Technická univerzita v Liberci
Hospodářská fakulta

Studijní program: M 6209 Systémové inženýrství a informatika
Studijní obor: Manažerská informatika

Vývoj e-business aplikace v jazyce Java

Developing an e-business application in Java

JANA JIRÁSKOVÁ

DP-MI-KIN-2009-15

Vedoucí práce: Ing. Vladimíra Zádová, Ph.D., katedra informatiky (KIN)
Konzultant: Ing. Vladimír Dvořák, Mailprofiler Development s.r.o.
Počet stran: 74 Počet příloh: 0
Datum odevzdání: 22. května 2009

Prohlášení

Byla jsem seznámena s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědoma povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracovala samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

V Liberci, 22. května 2009

Anotace

Tématem této diplomové práce je vývoj e-business aplikací v programovacím jazyce Java. V jejím rámci budou představeny a srovnány jednotlivé Java technologie, které se v současnosti používají k vývoji rozsáhlých systémů. Z porovnání budou vyvozeny závěry, některé technologie budou označeny za ustupující, jiné naopak za progresivní. Diplomová práce také poodkrývá problematiku objektového návrhu aplikací za pomoci metodiky vývoje softwaru Unified Software Development Process (USDP). Jejím cílem je ukázat vývoj konkrétní modelované e-business aplikace – konkrétně systému elektronického obchodního domu. Celá tato část bude prezentována v duchu metodiky USDP – od zkoumání požadavků a modelování případů užití, přes objektovou analýzu (s tvorbou statického a dynamického analytického modelu) až k samotnému pracovnímu postupu Návrh, v rámci kterého dojde k propojení s progresivními Java technologiemi.

Klíčová slova:

Java, Servlet, Java Server Pages, Java Server Faces, JBoss Seam, XHTML, expression language, objektové modelování, metodika Unified Software Development Process, požadavky, konverzace, UML, diagram případu užití, analýza, návrh, stavový diagram, sekvenční diagram, diagram nasazení, implementace, programování.

Annotations

Topic of my thesis is process of developing an e-business application in Java programming language. Different Java technologies (frequently used in these days) will be introduced and compared. And conclusions will be made - some technologies will be marked as archaistic, others will be marked as progressive. Object modeling will be introduced in thesis too (Unified Software Development Process methodology - USDP). Purpose of this part is to show how to develop an e-business application – system of electronic department house. Methods of USDP methodology will be used in this section – from modeling use case diagrams, via creating an analytical model (with dynamic and static views), to design. Progressive Java technologies and their usage will be also presented in Design section.

Keywords

Java, Servlet, Java Server Pages, Java Server Faces, JBoss Seam, XHTML, expression language, object modeling, methodology Unified Software Development Process, USDP, use case, conversation, UML, use case diagram, analysis, design, state diagram, sequence diagram, deployment diagram, implementation, programming.

Obsah

Seznam tabulek

Seznam obrázků

Seznam zkratek a symbolů

atd.	a tak dále
AWT	Abstract Windowing Toolkit
CGI	Common Gateway Interface
CRUD	Create, Reade, Update, Delete
GUI	Graphic User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JSF	Java Server Faces
JSP	Java Server Pages
JVM	Java Virtual Machina
tzn.	to znamená
UML	Unified Modeling Language
UP	Unified Process
USDP	Unified Software Development Process
XML	Extensible Markup Language

1 Úvod

S rozšířením povědomí a dostupnosti internetu se mění i požadavky na to, jak může firma obstát v tržním světě. Elektronické obchody, internetové bankovnictví – všechny tyto oblasti se pro zákazníka stávají samozřejmostí, nikoli „vymožeností“, bonusem, který ekonomickým subjektům přinese body navíc.

Hlavním cílem této diplomové práce je poodkrýt současný stav na poli vývoji e-business aplikací v jazyce Java. V jejím rámci budou srovnány jednotlivé technologie v současnosti využívané v e-business aplikacích. Cílem této části je dokázat tvrzení, že některé technologie jsou „zastaralé“ (nikoli nepoužitelné a nevyužívané!) a jiné progresivní.

K těmto závěrům jsem oprávněna dojít, protože mám za sebou několikaletou programátorskou praxi, v jejímž rámci jsem aktivně spolupracovala na vývoji dvou odlišných Java aplikací (a do světa dalších dvou jsem alespoň lehce nahlédla). A ze své praxe mám zkušenosti jak se „starým“, tak i s „novým“.

Tato diplomová práce se zabývá vývojem aplikací, které budou v konečné fázi nasazeny do prostředí internetu. V současné době dochází totiž k postupnému ústupu desktopových aplikací, které jsou dynamicky nahrazovány tenkými webovými klienty. To sebou ale přináší řadu úskalí a problémů. Nasazením aplikace do internetového prostředí ji totiž zpřístupňujeme doslova celému světu. A tím ji zároveň vystavujeme nespočtu potencionálních hrozeb a útočníků.

Internetové aplikace – ať už elektronické obchody, nebo o poznání složitější – totiž mnohokrát ke své funkčnosti potřebují uchovávat citlivá data (například o uživatelích). Tato data bývají dokonce chráněna zákony a jejich zneužití je trestné. Na vývojářích tedy leží odpovědnost takové informace řádně zabezpečit a zamezit nepovolaným osobám, aby k nim získali přístup. Tato problematika úzce souvisí s autentizací (identifikací uživatele) a autorizací (přidělením přístupových práv). Citlivá data přenášená po světové síti by také nikdy neměla putovat v nešifrované podobě. Ukázat, jak se tento problém řeší

v současnosti za pomocí moderních Java technologií, je jeden z cílů této práce.

Jazyk Java je navíc čistě objektově orientovaný. Proto vývoj za pomocí souvisejících technologií úzce souvisí s objektovým modelováním a objektovými metodikami vývoje a údržby aplikací. Dalším cílem je tedy i odhalit a zprůhlednit celý proces objektově orientovaného vývoje. S objektovým modelováním úzce souvisí jazyk UML. Ten sám ale nestačí, nutné je také zvolit konkrétní metodiku.

Objektová metodika samozřejmě není k dispozici jediná, nicméně já budu v rámci práce využívat metodiku Unified Softvare Development Process (USDP, zkráceně Unified process – UP). K tomuto rozhodnutí mě vede především to, že metodika UP je obecná (ale intuitivní) metoda pro vývoj softwaru, která nejvíce koresponduje s mými konkrétními zkušenostmi. Sama jsem totiž byla přítomna vývoji e-business aplikace v praxi. Konkrétně se jednalo o produkt na řízení marketingových kampaní – XCampaign, který je vyvíjen mým současným zaměstnavatelem – společností Mailprofiler Development. Do vývoje této aplikace jsem byla zapojena od samého počátku (v dubnu roku 2008) až do současnosti, kdy je aplikace komerčně nasazena a je využívána například částí evropské korporace společnosti Google.

V rámci praktické části bych pak přistoupila k modelování konkrétní e-business aplikace. Nejprve bych se chtěla zabývat požadavky a analýzou na systém, pro tyto účely jsem po zvážení vybrala systém obchodního domu (s nabídkou zboží v různých odděleních). A posléze bych v části návrhu ráda poukázala na to, jak lze využít současné moderní technologie, se kterými mám také zkušenosti z praxe (JSF a JBoss Seam) při vývoji takové aplikace.

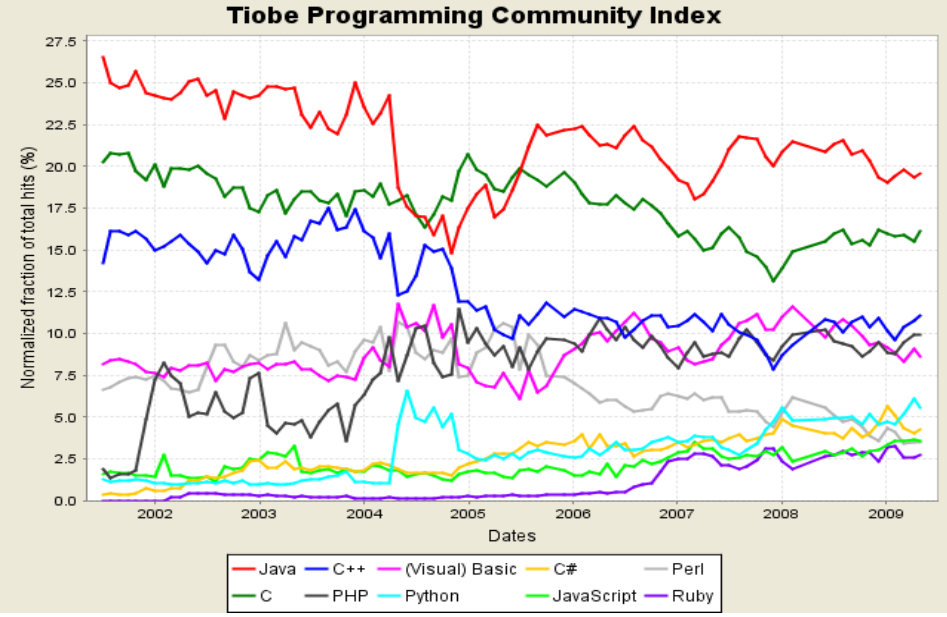
Cílem praktické části není kompletní zmapování a kompletní realizace aplikace obchodního domu. Mojí snahou bude podložení vlastních dovedností v oblasti analýzy objektově orientovaného systému a demonstrace využití progresivních Java technologií při tvorbě e-business aplikace.

2 Současný stav na poli vývoje webových aplikací v jazyce Java

Ve světě Javy se v průběhu vývoje objevilo již několik technologií, které řešily programování dynamických webových aplikací. A i v současnosti existuje celá škála Java technologií, která zajišťuje funkcionalitu e-business aplikací. Pojďme se tedy nejprve zaměřit na otázku, proč je právě Java v kombinaci s webem tak dynamicky rozvíjejícím se jazykem, který nabízí stále nové a nové možnosti.

2.1 Východiska rozvoje nových Java technologií

Programovací jazyk Java patří dlouhodobě mezi špičku ve vývoji aplikací. Toto tvrzení již několik let potvrzuje například vývoj TIOBE indexu.



Obr. 1: Vývoj TIOBE indexu mezi rokem 2001 a květnem roku 2009.

zdroj: http://www.tiobe.com

Tento index (aktualizovaný každý měsíc) sice přímo neměří, který programovací jazyk je v současnosti nejvyužívanější, či nejlepší, nicméně tuto skutečnost odráží alespoň nepřímo. Metodologie jeho měření je totiž založena na celosvětové dostupnosti vývojářů, pořádání zaměřených výukových kurzů a dostupnosti produktů v daném programovacím jazyce. A

právě Java v tomto měření dlouhodobě dominuje. [12]

Protože Java vývojáři jsou na trhu početně silně zastoupeni, a stejně tak realizovaných projektů v tomto programovacím jazyku není málo, proto stále vznikají nové požadavky na vývoj samotné Javy. Java se tak pod dynamickým tlakem stále mění - vznikají nové a nové technologie, které vylepšují její specifické funkce.

2.2 Common Gateway Interface - CGI

První technologií, která znamenala první krok v oblasti serverového programování – byla technologie CGI skriptů (Common Gateway Interface). Tato technologie sama o sobě není přímo vázaná na Javu – jedná se o samostatný projekt. Nicméně, jelikož technologie CGI skriptů přímo souvisí s problematikou Java servletů, je nutné se s ní trochu seznámit.

Webový server a CGI program komunikují na úrovni procesu operačního systému. Celá filozofie CGI vychází z předpokladu, že webový server je schopen uložit data uložená v požadavku GET do konkrétní proměnné prostředí. A CGI program pak pomocí této proměnné může zjistit obsah parametrů obsažených v URL adrese. [3]

Ve své době znamenalo CGI velký pokrok a bylo právem považováno za mocný nástroj. Tato technologie měla ale i několik zjevných nedostatků. Při zpracování požadavků klientů se spouštělo množství procesů, které nadměrně zatěžovaly jak server, tak procesor a značně tak komplikovaly generování stránek s odezvou. CGI skripty také nebyly schopny žít během více požadavků a bylo tedy nutné uchovávat některé stavové informace za pomocí souborů či databáze. [6]

2.3 Java servlety

Java servlet představuje jakousi mezivrstvu – program, který běží na webovém serveru a působí jako střední vrstva mezi požadavkem přicházejícím z webového prohlížeče a aplikací na serveru. Servlety byly skutečně účinnou odpovědí technologie Java na

programování pomocí CGI skriptů. [6]

Klasický servlet plní tyto úkoly:

1. Přečte všechna data zadaná uživatelem. Klasicky obsah webového formuláře (není ale vždy podmínkou).
2. Vyhledá všechny další informace o požadavku, které jsou uloženy v požadavku HTTP (včetně schopností prohlížeče, uložených cookies atd.).
3. Zpracuje výsledek. V této fázi může servlet například přistupovat k databázi atd.
4. Zformátuje výsledek uvnitř dokumentu, většinou do podoby HTML souboru.
5. Nastaví vhodné parametry HTTP odezvy. Tzn. například správnou hlavičku dokumentu, ve které předáváme informace o tom, jaký typ dokumentu se vrací klientovi, nastavení cookies atd.
6. Odesílání dokumentu zpět klientovi. [6]

Samotné programování Java servletů není příliš obtížné. Každý servlet musí dědit od třídy `HttpServlet` a pak pouze implementovat správně vhodné metody.

Důležité metody jsou `doGet` a `doPost`. Příslušná metoda je volána v případě, kdy je vyvolán http požadavek GET (volá se `doGet`), popř. POST (kdy se volá `doPost`). V rámci této dvojice metod je dobrým zvykem volat metodu `processRequest`, která připraví výstup – v příkladu HTML stránku s výstupem: „Hello World!“

Jako parametry jsou těmto metodám předávány objekty obalující HTTP požadavky a také mnoho dalších důležitých informací, se kterými pak můžeme pracovat a vytvářet mnohem složitější (a nepochybně účinnější) servlety. Můžeme například číst hodnoty HTTP parametrů, přistupovat k databázovým zdrojům atd.

```
import javax.servlet.*;
import javax.servlet.http.*;

public class HalloWoldServlet extends HttpServlet {

    // metoda vyvolána požadavkem GET
    protected void doGet(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, java.io.IOException {
        processRequest(req, res);
    }

    // metoda vyvolána požadavkem POST
    protected void doPost(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, java.io.IOException {
        processRequest(req, res);
    }

    // metoda volána v rámci metod doGet a doPost
    protected void processRequest(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, java.io.IOException {
        res.setContentType("text/html");
        java.io.PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hallo World</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("Hallo World!");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

2.4 Srovnání: CGI versus Java servlety

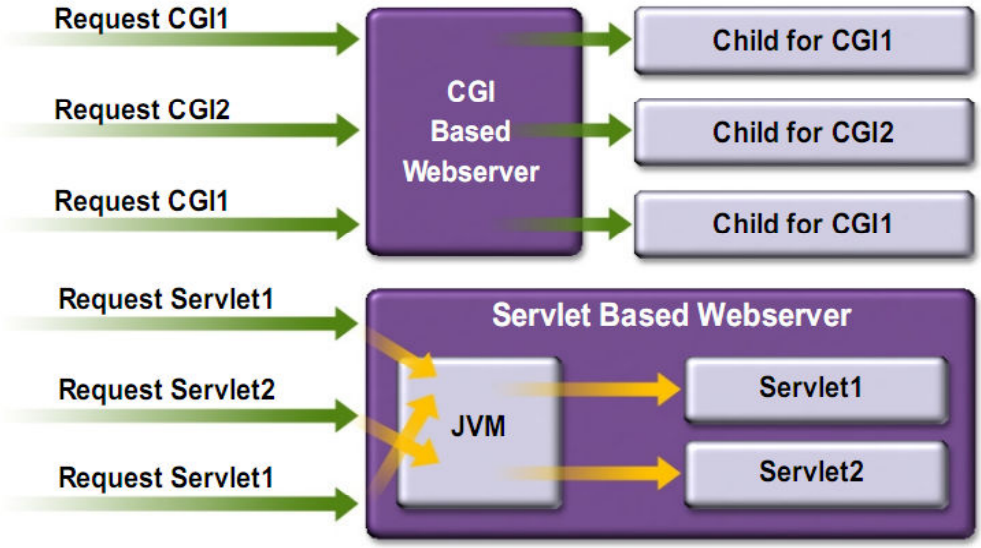
Technologie servletů vytvořila za pomocí Javy jakéhosi prostředníka mezi vrstvou, kterou pro komunikaci využívalo čisté CGI a samotnou aplikací. Servlety tedy nepochybně nabízejí mnohá vylepšení.

Porovnání CGI a Java servletů v jednotlivých sledovaných oblastech ukazuje tabulka č. 1.

Tab. 1: Porovnání servetů a CGI podle základních kritérií

Oblast	**Java servlety**	**CGI**
Výkonnost	Servlet je program. Jednou zkompilován a načten do paměti, potom už pouze odpovídá na klientské požadavky. Stejně tak může uchovávat informace z požadavku na požadavek bez nutnosti spouštění nových procesů. Servlety jsou navíc schopny sdílet data navzájem, což je vhodné pro optimalizaci celých procesů.	CGI spouští nový proces pro každý nově příchozí požadavek http (viz obrázek č. 6). Je-li požadavek krátký, může se dost dobře stát, že obstarávání požadavku bude trvat déle, než jeho samotné vykonávání, ani nemluvě o nárocích kladených na systém.
Programové vybavení	Servlet má za sebou bohaté aplikační programové rozhraní (API) jazyka Java, které může (až na grafické knihovny) plně využívat.	CGI programy jsou často vykonávány prostými příkazovými interprety operačních systémů.
Přenositelnost	Mohou komunikovat s webových serverem bez znalosti a použití specifického serverového API. Jsou tedy snadno přenositelné mezi různými typy aplikačních serverů.	Klasické CGI programy nemohou komunikovat s webovým serverem, rozhodně nikoli bez použití specifického serverového API. Tato vlastnost tak velmi komplikuje jejich přenositelnost.

Zdroj: HALL, M. Java servlety a stránky JSP [3].



Obr. 2: Technologie CGI versus Java servlety

Zdroj: http://mediacast.sun.com/users/MiKo22/media/JavaEE_servlets_CZ

Na obrázku č. 2 se ještě jednou vracím k problematice zpracování požadavků pomocí obou technologií. Zatímco technologie CGI není schopná uchovávat informace z požadavku na požadavek, je nucena pro každý zvlášť vytvořit vlastní proces. Oproti tomu jsou servlety v prostředí webového serveru jednou zkompilovány a od té chvíle zůstávají dostupné. Virtuální stroj Javy (Java Virtual Machine – JVM) pak pouze směruje konkrétní požadavky konkrétním servletům.

Můžeme tedy jasně konstatovat, že oproti CGI přináší technologie servletů samá zlepšení. Sama má však stále mnoho nedostatků (hlavně pak tvorbu prezentační vrstvy v rámci Java kódu). Tyto a další nevýhody se pokusila vyřešit další technologie – Java Server Pages (JSP).

2.5 JSP

Technologie JSP se snažila potlačit největší nevýhodu Java Servletů, kterou byla tvorba prezentační vrstvy uvnitř Java kódu. Tento postup rozhodně nebyl praktický, když uvážíme, že design aplikace mají obvykle na starosti designéři, nikoli vývojáři. A ty dost často (není to ani z jejich pozice žádoucí) nemají znalosti jazyka Java. Proto je jakékoli

stylování výstupů tvořící servlety organizačně obtížné.

Na rozdíl od servletů je JSP stránka podobná stránce v jazyce HTML. Liší se od ní pouze přítomností zvláštních syntaktických elementů, díky kterým je schopna dynamicky měnit svůj obsah. Technologie JSP umožňuje také účinné kombinování HTML elementů přímo s jazykem Java, díky kterému vnukneme stránce její dynamický obsah.

Zpracování JSP stránek probíhá rozdílně od statických HTML dokumentů. Podle specifikace JSP se proces na straně serveru, který řídí zpracování JSP stránek, nazývá JSP kontejner. JSP kontejner překládá JSP stránky do kódu Java servetů a výsledek poté překládá a načítá do servlet kontejnerů. [3]

Ačkoli tedy programátor neprogramuje přímo Java servlety (už žádné metody `doGet` a `doPost`) a operuje v prostředí klasického HTML promíchaného s prvky jazyka Java, po zpracování není JSP stránka nic jiného, než právě Java servlet. Psát JSP je ale o mnoho snadnější a průhlednější, než tvořit servlety.

Celý iniciální požadavek na konkrétní JSP stránku probíhá v následujících pěti krocích:

1. Interpretuje se JSP stránka.
2. Vygeneruje se Java servlet.
3. Servlet se pomocí standardního překladače, který je dodán s JSP kontejnerem, převede do bajtového kódu Java.
4. Servlet je načten do virtuálního stroje Java servlet (JVM) kontejneru.
5. U servletu je vyvolána služební metoda. [3]

Požaduje-li prohlížeč následně stejnou JSP stránku, a nebyla-li stránka od posledního volání změněna, musí JSP kontejner provést znovu pouze krok pět. Pokud se však stránka změnila, je nutné zopakovat všech pět kroků. Opětovné volání stejné JSP tedy probíhá

rychleji, než iniciální žádost o ni. A to z důvodu, který byl zmíněn již v rámci kapitoly porovnávající CGI a Java servletů na obrázku č. 2. Jakmile je servlet jednou vygenerovaný, žije v rámci prostředí webového serveru a pouze odpovídá na příchozí požadavky.

2.6 Srovnání: JSP versus Java servlety

Stejně jako nástup servletů oproti CGI znamenal velký krok dopředu, stejně tak technologie JSP znamenala obrovský posun (a velké zjednodušení) oproti čistým servetům.

Se stejnými kritérii porovnání si tu ale nevystačíme. Obě technologie jsou plně přenositelné, obě mohou využívat bohatého API jazyka Java a obě jsou poměrně výkonné. JSP však hlavně řeší problematickou tvorbu prezentační vrstvy u servletů. Je jistě o moc jednodušší a přehlednější psát (či posléze editovat) čisté HTML tagy, nikoli `println` řádky programového kódu, které obsluhovaly generování výstupu v čistých servletech.

Design takové aplikace je také mnohem průhlednější. A tak oddělení designerských a vývojářských rolí v rámci vývojového týmu není již za použití technologie JSP nijak problematické.

Stejně tak je JSP stránka schopna plnit jakékoli funkce, které plnil klasický servlet – jako je například získávání hodnot parametrů požadavku. Proto ji lze označit za úspěšného nástupce technologie servletů. Stále jí však něco schází. Sama o sobě totiž nenabízí žádný propracovaný komponentový model. Vytváření formulářů, tabulek – celé prezentační vrstvy, je tu i nadále ponecháno čistě na HTML, které samo o sobě není ideální.

Co je ale ještě horší, že JSP technologie umožňuje volně míchat HTML kód prezentační vrstvy a Java kód vrstvy aplikační (celá JSP je překládána na servlet = speciální Java třídu, Java kód uvnitř jí tedy nedělá žádné potíže). Je tak tedy velmi jednoduché narušit třívrstvou architekturu aplikace. Jinými slovy – jako Java servlety nutily vývojáře vkládat

prezentační logiku do aplikační vrstvy, JSP umožňuje (nikoli vynucuje!) vkládat aplikační logiku do vrstvy prezentační.

2.7 JSF (JavaServer Faces)

Nedostatečný komponentový model JSP se snažila řešit celá řada volně dostupných i komerčních technologií, například projekt Jakarta Struts. A právě ze snahy vylepšit tento projekt nakonec vzešla technologie Java Server Faces (JSF), která je od verze 1.5 součástí enterprise verze programovacího jazyka Java. Stejně jako Swing nebo AWT (javovské komponentové knihovny pro grafické uživatelské rozhraní – GUI) je JSF základní komponentovou knihovnou pro rozhraní webových aplikací.

Celý chod JSF aplikace je řízen událostmi. Toto řízení definuje programátor ve třídách listenerů (podobně jak je tomu při programování GUI rozhraní). JSF podporuje dvě události: ActionEvent a ValueChangedEvent. ActionEvent nastává když uživatel odešle formulář nebo klikne na tlačítko, ValueChangedEvent když se hodnota v JSF komponentě změní. Kdykoli uživatel něco udělá, například stiskne tlačítko, nastane událost. Oznámení o události je poslána přes HTTP server. Na serveru je webový kontejner, který zaměstnává speciální Faces servlet. [5]

Dalším důležitým objektem je javax.faces.context. FacesContext zapouzdřuje všechny důležité informace o požadavku klienta.

JSF nabízí standardní komponentu pro každé vstupní pole, které je součástí jazyka HTML. Mimo to nabízí možnost tvorby svých vlastních komponent, nebo skládání klasických komponent do příbuzných shluků.

JSF také nabízí snadnou serverovou validaci, možnost konverze dat, kontrolu nad navigací v rámci aplikace atd. Je také od svého základu tvořeno jako rozšiřitelné a konfigurovatelné.

Na trhu je k dispozici mnoho implementací, které rozšiřují jeho základní standardní funkce.

Každá aplikace využívající komponent JSF, musí být správně nakonfigurována. Ke konfiguraci se využívají XML soubory (především web.xml a faces-config.xml).

K volání příslušných listenerů, či mapování objektů prezentační vrstvy, využívá JSF takzvaný „expression language“, nikoli Javu přímo. Expression language odkazuje na typ java tříd, tvz. „backing bean“ (termíny expession language a backing bean se ujaly mezi českou odbornou komunitou v originální anglické podobě). Aby byla daná backing bean viditelná v rámci prezentační vrstvy, musí se registrovat právě pomocí XML (faces-config.xml). Pomocí XML definujeme jméno, pod kterým bude vystupovat a kontext platnosti, například session, nebo request. V rámci stránky pak vyvoláváme jednotlivé metody právě pomocí definovaného jména v rámci řetězce Expression language (ohraničený #{ … }).

```
<h:form>
<h:outputLabel value="First Number" for="firstNumber" />
<h:inputText id="firstNumber" value="#{CalcBean.firstNumber}"
required="true" />
<h:message for="firstNumber" />

<h:outputLabel value="Second Number" for="secondNumber" />
<h:inputText id="secondNumber" value="#{CalcBean.secondNumber}"
required="true" />
<h:message for="secondNumber" />

<h:commandButton id="submitAdd" action="#{CalcBean.add}"  value="Add" />
<h:commandButton id="submitMultiply" action="#{CalcBean.multiply}"
        value="Multiply" />
</h:form>
```

Metody a vázané atributy v třídě reprezentující backing bean by mohly vypadat následovně.

```
private int firstNumber;
private int secondNumber;

public String add() {
  result = calculator.add(firstNumber, secondNumber);
  return "success";
```

```
}

public String multiply() {
  result = calculator.multiply(firstNumber, secondNumber);
  return "success";
}
```

A proč dané metody vrací řetězec „success“? V tomto případě se nejedná o žádné klíčové slovo. Rozhodnutí o tom, co mají metody backing bean vracet v případě úspěchu záleží pouze na rozhodnutí vývojového týmu. Hodnota tohoto návratového řetězce (ať už je jakákoli) má ale ohromný význam při definování navigačních pravidel pro danou aplikaci. Tj. pravidel, co se stane s aplikací, pokud uživatel správně zadá dvojici vstupů? Kam se přesune? A co se stane v opačném případě? Právě toto musíme také definovat v konfiguračním souboru faces-config.xml a to následujícím způsobem:

```
<navigation-rule>
  <from-view-id>/calculator.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/results.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

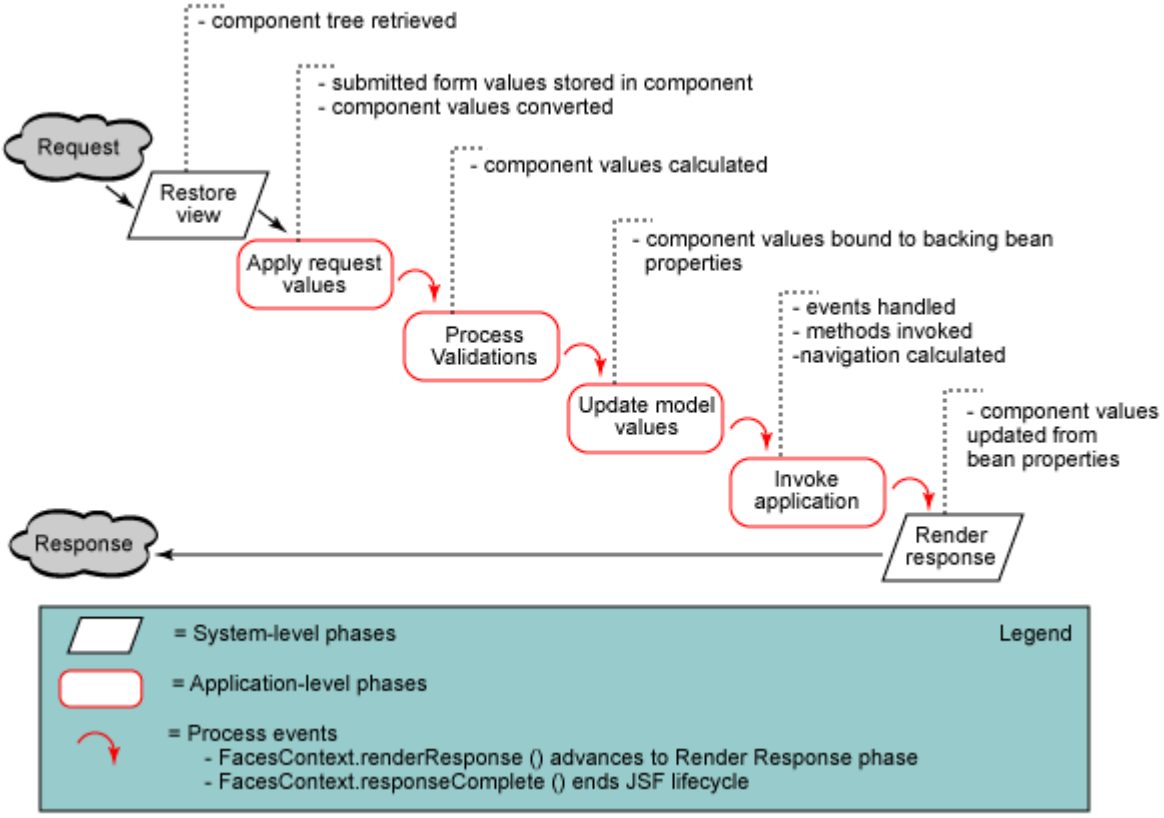
Toto navigační pravidlo si můžeme přeložit následovně: „Pokud operace proběhne úspěšně (metoda v tomto případě vrací řetězec „success“), aplikace se přesměruje na stránku `result.jsp`.

Základ pro použití technologie JSF je tedy poměrně jednoduchý. Programátor sám nemusí vědět příliš o tom, co se děje na pozadí, a může vytvářet funkčně bohatou aplikaci a to poměrně v krátkém čase. Využití JSF totiž opravdu šetří spoustu řádek kódu. Protože samo nabízí možnosti validace, konverze zadaných hodnot a mnohé další. Problémy, pro které byly tedy dříve potřebné další a další metody, si nyní technologie JSF řeší sama. A programátor se tedy může spolehnout na to, že mu do aplikační vrstvy přichází validní data.

Co se ale děje v pozadí, a jak je vůbec možné, že se programátor správně využívající komponent JSF může spolehnout na správnost svých aplikačních dat? Všechno se objasní, podíváme-li se na problematiku životního cyklu každé JSF aplikace. Ten se totiž skládá

z následujících fází:

1. Načtení / Obnova pohledu (restore view).
2. Zaznamenání hodnot požadavku (apply request values).
3. Proces validace (process validations)
4. Aktualizace hodnot v modelu (update model values)
5. Volání aplikace (invoke application)
6. Předložení odpovědi (render response) [8]



Obr. 3: Životní cyklus JSF

Zdroj: HIGHTOWER, R. JSF for nonbelievers: The JSF application Life cycle [8]

Celý životní cyklus aplikace není neměnný. Programově můžeme fáze vyvolat předčasně, nebo celý cyklus ukončit. Ve většině případů jej ale ponecháme právě tak, jak byl navržen a to z jednoduchého důvodu. Všech šest fází má svou vlastní logiku a souvislost. Většinou bude tedy při vývoji aplikace životní cyklus JSF „spojencem“ vývojářů, nikoli něčím, co je

potřeba měnit.

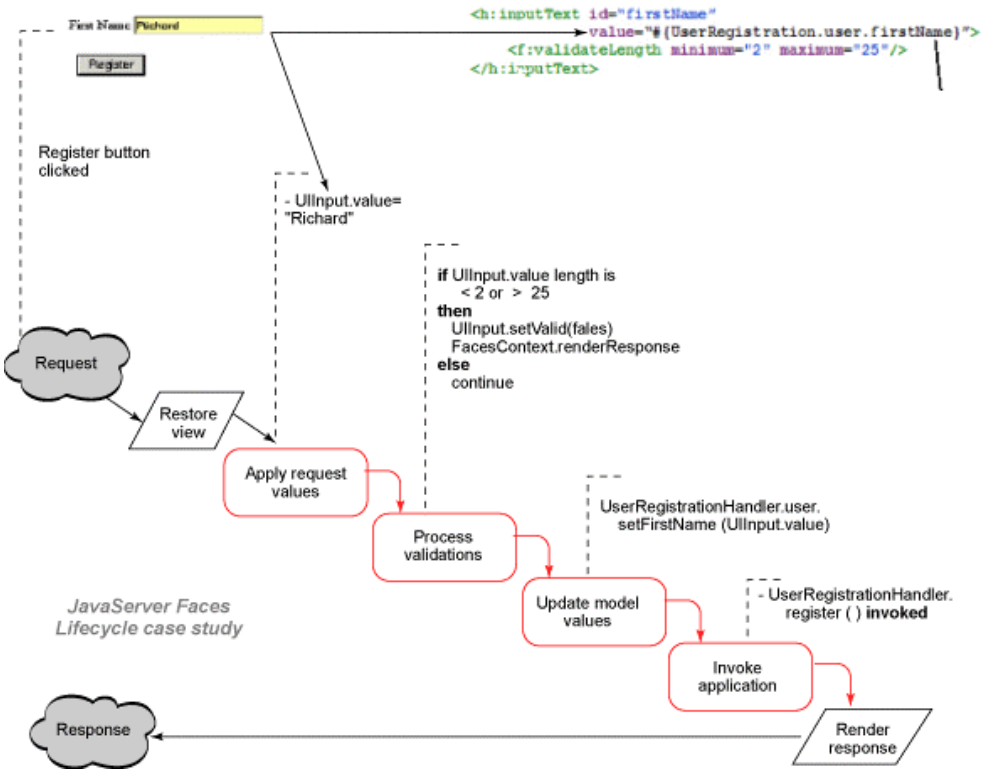
V první fázi životního cyklu aplikace se vykreslí náhled požadovaného view (např. s JSF formulářem, jež čeká na interakci s uživatelem). Ve chvíli, kdy uživatel vyplní formulář a odešle jej, hodnoty se nejprve uloží do samotné komponenty a dojde ke všem požadovaným konverzím. JSF například umožňuje přímou konverzi řetězců do číselných formátů. Již v této fázi by tedy v případě, že by daná položka nešla zkonvertovat, došlo k přerušení celého životního cyklu a nesprávná hodnota se ještě ani nepřiblížila k tomu, aby ovlivnila vnitřní aplikační logiku aplikace. [5]

Pokud konverze proběhne úspěšně, dostáváme se do další fáze v jejímž rámci dojde k ověření, že dané hodnoty odpovídají definovaným validačním pravidlům (opět, JSF nabízí spoustu standardních validátorů, u číslic maximální a minimální hodnodu, u řetězců například maximální délku řetězce atd.). Opět, pokud zde dojde k nějaké chybě, vše je včas zachyceno bez toho, aniž by to ovlivnilo aplikaci. V další fázi, kdy už se můžeme spolehnout na to, že data poskytnutá uživatelem jsou validní, se tato data uloží do objektu a následně s nimi můžeme pracovat v nitru aplikace. V předposlední fázi životního cyklu jsou vyhodnoceny návratové hodnoty metod třídy a na jejich základě jsou vyhodnocena navigační pravidla. Poslední fáze životního cyklu pak reaguje zobrazením výstupu odpovídajícímu danému navigačnímu pravidlu.

Pojďme se chvíli ještě pozastavit nad validací a konverzí. JSF, jak už jsem zmínila výše, nabízí sadu základních validátorů a konvertovacích operací. Jedna z nejjednodušších možností je využít právě je.

```
<h:inputText id="age" value="#{UserRegistration.user.age}">
        <f:converter id="javax.faces.Short"/>
        <f:validateLongRange maximum="150" minimum="0"/>
</h:inputText>
```

Na obrázku č. 4 můžeme názorně sledovat, co bylo popsáno výše v rámci životního cyklu aplikace.



Obr. 4: Zpracování odeslaného formuláře v rámci JSF životního cyklu.

zdroj: HIGHTOWER, R. JSF for nonbelievers: JSF conversion and validation [7]

JSF také nijak nebrání vytváření vlastních složitějších konvertorů a validačních pravidel. Pokud se k tomuto kroku rozhodneme, stačí nám naprogramovat třídy implementující rozhraní Validator, resp. Converter a registrovat příslušné třídy v konfiguračním souboru faces-config.xml.

2.7.1 Facelets jako alternativa JSP při využití JSF

JSF není samostatná technologie, nýbrž technologie závislá. Ke své prezentaci potřebuje využít jiných nástrojů. Buď již výše zmiňované technologie JSP, nebo technologie Facelets.

Technologie Facelets a je založena na budování tvz. stromu komponent. A na rozdíl od JSP, návrh Facelets byl již od počátků spjatý s teorií o životním cyklu JSF aplikace. [9] Facelets jsou především jednoduchou šablonovací technologii. Základní šablona facelets může vypadat následovně:

```
<div id="header">
    <ui:insert name="header">
        <ui:include src="header.xhtml"/>
    </ui:insert>
</div>
<div id="content">
    <ui:insert name="content">
        <ui:include src="content.xhtml"/>
    </ui:insert>
</div>
```

Pomocí tohoto kódu docílíme toho, že se obsah stránky header.xhtml uloží jako implicitní obsah bloku header. Content.xhtml se stejně tak nastaví jako základní obsah pro blok content.

Tuto šablonu mohou následně využívat ostatní stránky, které v naší aplikaci využíváme. A ještě více. Pokud potřebujeme v takové stránce předefinovat pouze tělo stránky a hlavičku ponechat stejnou, nemusíme měnit implicitní obsah hlavičky definovaný v šabloně. Tím nám Facelets dávají velké možnosti ve vytváření unifikovaných aplikací, bez sebemenší nutnosti opakování části kódu.

```
<ui:composition template="template.xhtml">
  <ui:define name="content">
        <h:form id="form">
                …
        </h:form>
  </ui:define>
<ui:composition>
```

2.8 Srovnání: JSP versus JSF

V této kapitole bude zpracován pohled na to, jak (a zdali) technologie JSF reaguje na slabé stránky samotně používané technologie JSP (absence vlastní komponentové knihovny a možnost vkládání aplikační logiky do prezentační vrstvy). A to ve dvou případech – použití JSF komponent v rámci JSP, či jejich vkládání do šablon Facelets.

2.8.1 Absence vlastní komponentové knihovny

- **JSF + JSP.** JSF je komponentovou knihovnou, kterou můžeme zcela volně využívat v prostředí JSP. Tuto slabinu JSF v kombinaci s JSP tedy nepochybně řeší. JSP stránky mají ale už ze své podstaty několik omezení. Jsou při svém prvním načtení překládány na servlet a tento servlet se aktualizuje znovu generuje pouze v případě, že došlo k nějaké změně uvnitř java kódu stránky. JSF komponenty však nejsou součástí tohoto kódu, v očích JSP technologie se jedná pouze o klasické elementy s vlastním jmenným prostorem. Pokud tedy třeba dojde ke změně nějakého atributu v rámci komponenty JSF, servlet nezaregistruje potřebu znovu se vygenerovat.

- **JSF + Facelets.** Na rozdíl od JSP, stránka psaná za pomocí technologie Facelets není překládána na Java servlet. Místo toho je transformována do stromu komponent, který je schopen reagovat na provedené změny a aktualizovat kdykoli svůj stav. V rámci Facelets navíc lze využít šablonovací nástroje této technologie (a vytvářet tak společná záhlaví, zápatí stránek bez jejich opětovného vkládání). I tyto společně definované části ve Facelets šabloně mohou obsahovat jednotlivé komponenty JSF, které můžou být napojeny na jednotnou aplikační logiku.

2.8.2 Vkládání aplikační logiky do prezentační vrstvy

- **JSF + JSP.** Tuto slabinu technologie JSP komponentová knihovna JSF sama o sobě vůbec neřeší. JSP – ačkoli doplněna o komponenty z dílny JSF – stále funguje tak, jak byla předurčena a její filozofie zůstává zcela nezměněna. Stále je tak postavena na možnosti míchání klasického HTML (nově navíc i JSF tagů) a programového kódu jazyka Java.

- **JSF + Facelets.** Míchat aplikační logiku s prezenční vrstvou je při použití Facelets v podstatě znemožněno. Soubory Facelets jsou totiž klasickými XHTML soubory, které přítomnost Java kódu rozhodně vylučují. Veškerou aplikační logiku tak můžeme do prezentační vrstvy napojovat pouze za pomocí expression language.

A jaký je tedy závěrečný soud nad technologií JSF? Jedná se nepochybně o velmi mocný nástroj, který programátorům nabízí mnoho osvědčených a hojně využívaných funkcí. Umožňuje skutečně snadné škálování aplikace do jednotlivých vrstev a vede programátory k tomu, aby vytvářely znovupoužitelný kód. I samotný základ – událostmi řízená architektura ve webovém rozhraní je v jazyce Java poměrně průlomová (doposud existovalo propracované událostmi řízené rozhraní jen pro GUI aplikace).

Nicméně JSF má i několik nevýhod. Vyžaduje například mnoho konfigurací v XML souboru. Což není problém, pokud je aplikace malá, nicméně pro rozsáhlé systémy je registrování každého navigačního pravidla, backing bean, konvertorů a validátorů, trochu nepřehledné.

JSF je pro Javu opravdu krokem vpřed. Nicméně i tak je spíše určeno pro další implementaci, nikoli pro samostatné využití. Takovou implementací můžou být například knihovny JBoss Richfaces, Icefaces a další (pro podrobnější seznámení a porovnání vlastností těchto implementací odkazuji na http://www.jsfmatrix.net/).

2.9 JBoss Seam

JBoss Seam je takzvaný integrační framework, který se snaží řešit některé nedostatky JSF. V současnosti se při vývojích aplikací běžně využívá řada frameworků. Za zmínku stojí například tyto (všechno technologie Java): Hibernate, Spring, Struts a další. Frameworky platí v praxi programátora nepochybně za užitečného pomocníka. Spojování vícera z nich v rámci jedné aplikace ale nezadržitelně vede k vytváření speciálních programových „lepidel“, které s aplikační logikou dané aplikace nemají mnoho společného. Jedním z progresivních řešení, které lze v současné době využít v rámci vývoje Java aplikací, je právě integrační framework Jboss Seam.

Svůj úspěch staví Seam především na anotacích. Odbourává nutnost programování tradičního controlleru z MVC návrhového vzoru. A to tak, že JSF stránkám v prezentační

rovině umožňuje využívat místo klasických JSF backing bean přímo entity. Veškerá potřebná konfigurace je řešena právě za pomocí anotací. [10]

Anotacemi Seam tedy téměř zcela odbourává nutnost konfigurace pomocí XML. Povětšinou XML zcela nezavrhuje, pokud tato možnost někomu vyhovuje, má tak volnou ruku v ní pokračovat. Nicméně u velmi rozsáhlých projektů je XML konfigurace (navigační pravidla, viditelnost komponent) časem s jistotou nepřehledná a těžko spravovatelná.

Jedny z nejpoužívanějších anotací, které Jboss Seam nabízí jsou:

- @Name. Jedná se o anotaci zdrojové třídy. Určuje jméno komponenty, pod kterým bude přítomna v kontextu.
- @Scope. Také anotace zdrojové třídy. Určuje kontext, ve kterém bude komponenta přežívat. [1]

V čem je Seam opravdu silný, to jsou právě různé úrovně kontextů, ve kterých mohou komponenty žít. K dispozici jsou následující:

1. Stateless – bezestavové komponenty.
2. Event – analogický k požadavku.
3. Page – definuje platnost komponenty vztaženou ke stránce.
4. Conversation – jádro Seamu, konverzace = série requestů za účelem předem daného cíle.
5. Session – analogie platnosti HTTP session.
6. Business process – tento kontext dlouží pro dlouhotrvající business procesy a může být rozprostřen dokonce mezi více uživatelů.
7. Application – globální kontext držící statické informace. [10]

Podrobněji bych se chtěla zabývat především jádrem kontextů Seamu, tj. konverzačním kontextem.

Konverzací je chápána souslednost několika HTTP požadavků, které v konečném důsledku tvoří celek, který sleduje nějaký cíl. Takovým cílem může být například dokončení registrace nového uživatele. Je to tedy právě konverzační kontext, který ve frameworku Jboss Seam řeší jednou pro vždy definitivně řeší otázku, jak nakládat s proměnnými mezi jednotlivými HTTP požadavky (například při vyplňování registračních průvodců vedoucích přes více stránek atd.).

I konverzace může být řízena pomocí anotací. A to konkrétně:

- @Begin. Anotace je cílená na metodu. Pokud je volána metoda označená touto anotací, je to povel pro Seam převést dočasnou konverzaci na konverzaci dlouhodobou (long running) nebo vnořenou (nested).
- @End. Taktéž tato anotace je umístěná nad deklarací metody. A pokud je taková metoda volána, je to povel pro Seam ukončit probíhající dlouhodobou, či vnořenou, konverzaci. [1]

V jednom prohlížeči je navíc možné na více záložkách vést v téže webové aplikaci více konverzací. Můžeme tedy zapomenout problematiku různého chápání webových záložek v různých prohlížečích. Pokud využijeme Jboss Seam a teorii o konverzačním kontextu máme jednotlivé záložky plně pod kontrolou.

Seam také zdokonaluje životní cyklus JSF, především pak jeho iniciální podobu. Pokud totiž budeme uvažovat nad životním cyklem JSF v situaci prvního načtení stránky, musíme konstatovat, že je poměrně „bezzubý“. Seam ale oproti tomu umožňuje napojit na každou stránku akce. Tyto akce jsou pak vykonány v průběhu inicializace. [1]

Dalším bodem, kde Seam povyšuje práci s JSF o stupeň výš, je validace vstupních dat.

Samotné JSF totiž podporuje validace, nicméně – v trochu zdlouhavé podobě. Obzvlášť pokud si uvědomíme, že klasické entity, vygenerované z databáze obsahují vlastní validační pravidla odpovídající databázovým požadavkům (např. Požadavky na délku, minimální a maximální hodnotu a další). Proč je tedy znovu přepisovat do JSF formulářů? Proto Seam přichází s jednoduchým řešením, jak validitu formulářů ověřit oproti omezením definovaným v entitách. A to pomocí tagů `s:validete`, popř. `s:validateAll`.

```
<h:inputText value="#{person.age}">
     <s:validate />
</h:inputText>

<s:validateAll>
     <h:inputText value="#{person.age}" />
     <h:inputText value="#{person.height}" />
</s:validateAll>
```

Seam také podstatně ulehčuje tvorbu takzvaných CRUD operací (create, read, update, delete), které jsou základem prakticky každé business aplikace. Obvykle programátoři řeší tyto operace pomocí návrhového vzoru Data Access Objects (DAO) návrhového vzoru. Protože ale mají všechny DAO objekty stejný základ, obsahuje Seam ve svém nitru vlastní CRUD framework s předpřipravenými DAO objekty, které lze velmi snadno implementovat (a to děděním od třídy EntityHome, která je součástí Seamu). [10]

Seam také rozšiřuje možnosti expression language. Na rozdíl od JSF, v rámci Seam aplikace můžeme pomocí EL volat libovolné metody (nikoli pouze speciální listenery s povinnou deklarací) a předávat jim libovolné parametry (což v čistém JSF nebylo možné). Seam navíc v rámci svých knihoven obsahuje i knihovny pro snadnou tvorbu PDF souborů a tvorbu emailů. A jako třešnička na dortu je pro účely autorizace k dispozici tzv. koncept Role-Based Security, díky kterému lze snadno docílit zabezpečení aplikace na několika úrovních. [1]

2.10 Srovnání: Čisté JSF versus JSF + JBoss Seam

Již z výkladu o frameworku Seam lze usoudit, že hlavním cílem je vylepšit „vady“ čistého JSF. A nutno podotknout, že v této oblasti je Seam poměrně úspěšný.

V rámci tabulky č. 2 jsou přehledně srovnány technologie čistého JSF a JSF v kombinaci s frameworkem Seam. Důraz je kladen na trojici porovnávaných kritérií (životní cyklus, možnosti využití expression language a konfigurace aplikace).

Mimo zmiňovaná kritéria se na stranu využití frameworku Seam přikláním také hlavně z důvodu existence kontextu Conversation, který v samotném JSF přítomen není.

Tab. 2: Srovnání použití čistého JSF oproti kombinaci s frameworkem Seam

Problematika	**Čisté JSF**	**JSF + JBoss Seam**
Životní cyklus JSF	Iniciální značně omezený, při opakovaném požadavku klasických sedm fází.	Klasický JSF cyklus obohacuje o akce a parametry spojené s konkrétními stránkami, tím lze účinně rozšířit iniciální podobu JSF životního cyklu.
Expression language	Neumožňuje volat libovolné metody, pouze tvz. listenery s předdefinovanými parametry ve třídě reprezentující „backing bean“.	Možnost volat jakékoli metody uvnitř definovaných komponent a možnost předávat jim libovolné parametry.
Konfigurace	Pomocí XML.	Pomocí Java anotací (či XML).

Zdroj: vlastní.

3 Modelování systému obchodního domu

Ačkoli netvrdím, že technologie servletů a JSP bychom měly přehlížet (jsou totiž v mnoha dnes úspěšných aplikacích využívány), když přistupujeme k návrhu zcela nové aplikace, je vhodné využít technologie, které nabízejí bohatší a kvalitnější škálu vlastností. Takové, které jsou v současnosti progresivní.

Ze srovnávacích kapitol (2.6, 2.8 a 2.10) vzešlo vítězně použití aplikačního frameworku JBoss Seam, který šikovně obaluje ostatní technologie (jako JSF, Hibernate pro přístup k databázím atd). Cílem této kapitoli je tedy nastínit postup při modelování a samotném programování aplikace právě za použití Seamu. Jako vzorová e-business aplikace bude sloužit systém elektronického obchodního domu.

Elektronický obchodní dům je prezentován jako sdružení elektronických obchodů. Každý z obchodů v rámci obchodního domu nabízí zboží ze své oblasti (některý může nabízet bílé zboží, jiný naopak knihy atd.). Administrace celého domu by ale měla být jednotná.

Protože v rámci této kapitoly bude dodržována posloupnost jednotlivých pracovních postupů metodiky UP, je potřeba se s ní nejprve seznámit.

3.1 Objektové modelování a metodika USDP (Unified Software Development Process)

Informace pro tuto kapitolu jsem čerpala především z literatury, a to hlavně knihy UML a unifikovaný proces vývoje aplikací autorů J. Arlowa a I. Neustadta [2].

Metodika USDP (nebo zkráceně UP) reprezentuje jeden z klasických postupů procesu tvorby softwaru. Vznikla v roce 1994 a na jejím vzniku se podíleli stejní lidé, kteří stojí i za jazykem UML (řeč je především o Ivaru Jacobsonovi).

Jazyk UML v současnosti dominuje na poli výrazových prostředků při analýze a návrhu informačních systémů. Sám však ale nenabízí žádnou metodiku, žádné konkrétní postupy řešení při takovém návrhu. Kvůli tomu přichází na scénu metodika UP. Zatímco UML je jazykovou částí projektu, procesní částí je metodika UP.

Metodika UP převzala základní myšlenku z metody Ericsson (Ericsson approach, 1967). Touto myšlenkou je, že velké systémy by se měly vytvářet postupně (přírůstkově) – rozdělením do jednotlivých stavebních bloků. Tento proces je znám také pod pojmem iterační vývoj projektu.

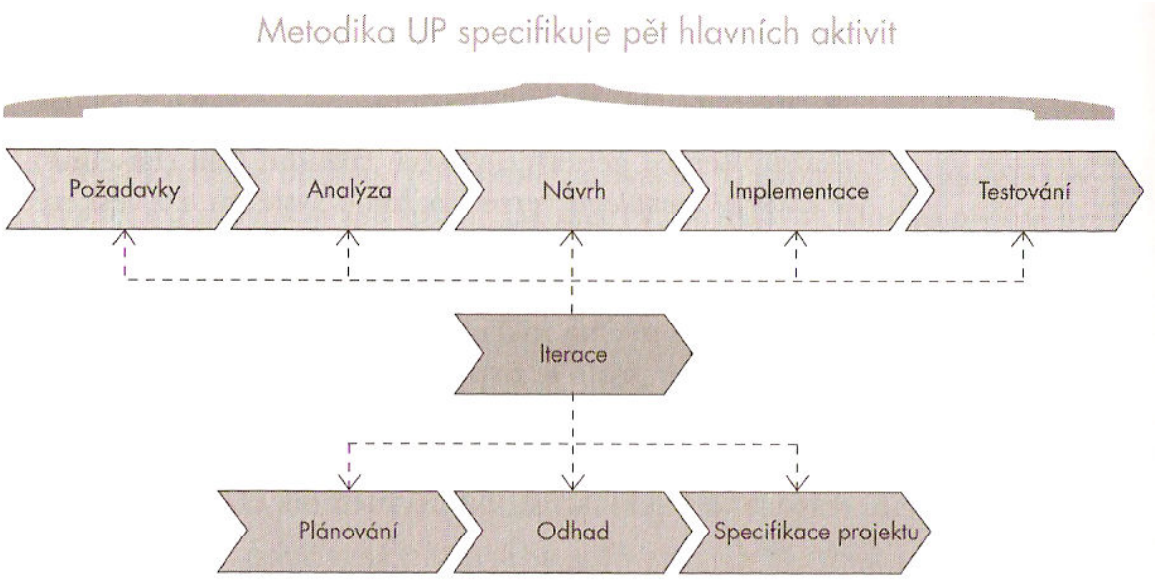
Další metodika, ze které autoři UP vycházeli, je Rational Objectory Process (1996). Tato metodika se skládala z několika šablon pro různé typy vývojových projektů softwaru. Autoři metodiky UP si totiž správně uvědomovali, že každý projekt je unikátní. Na jedné straně tak před nimi stála snaha odhalit určitá společná specifika při vývoji a realizaci projektů. Na straně druhé tu bylo ale vědomí, že vytvoření zcela univerzální metodiky je neuskutečnitelné (a někdy dokonce nežádoucí). Proto metodika UP vznikla hlavně intuitivní obecný princip založený na nejlepších doposud známých principech.

Jedním ze základních pojmů metodiky UP je iterační přístup. Ten vystihuje nutnost rozdělení rozsáhlejších projektů na dílčí části, jež je možné řešit samostatně. Základní myšlenka tak vychází z jasného předpokladu, že malé problémy se každému řeší lépe než ty rozsáhlé. Tyto samostatné menší projekty jsou pak v rámci metodiky považovány za iterace.

Vývoj každé iterace probíhá po vlastní základní linii. Výsledkem každé této vývojové linie by měla být část kompletní verze vyvíjeného systému, včetně veškeré přidružené projektové dokumentace. Iterace a jejich základní linie jsou vrstveny tak dlouho, dokud není dosažena finální podoba vytvářeného systému.

Rozklad projektu na sled iterací také umožňuje jistou dávku flexibility v procesu plánování. Nejjednodušší je samozřejmě sestavit časovou posloupnost iterací tak, že dokončení jedné vede k zahájení další. Není to však vždy nutně žádoucí. Někdy je výhodné realizovat jednotlivé iterace souběžně. Proto je nutné porozumět jednotlivým závislostem mezi artefakty každé iterace, což vyžaduje způsob vývoje softwaru založený na architektuře a modelování. Výhodou při souběžném zpracování iterací je možnost zkrácení celého projektu. Nejednou i lepší využití vývojového týmu. Podstatou je ale nikdy nepodcenit fázi plánování.

Základní linii každé iterace tvoří z pravidla pět základních pracovních postupů. Ty určují, co je třeba udělat, a jakým způsobem toho možné dosáhnout.



Obr. 5: Aktivity v rámci metodiky UP

zdroj: ARLOW, J., Neustat, I. UML a unifikovaný proces vývoje aplikací [2]

1. **Požadavky.** Zachycují to, co by měl systém dělat (nikoli jak toho bude dosaženo). Jedním z možných konečných výstupů pracovního postupu Požadavky je model případů užití. V rámci tvorby tohoto modelu dochází k hledání účastníků (osob – rolí – které s navrhovaným systémem budou pracovat). Patří sem také identifikace hranic systému (je nutné si správně ujasnit, jaké situace je potřeba v rámci systému řešit a jaké problémy už naopak spadají za jeho hranice). Opomenout se

samozřejmě nesmí ani samotné nalezení případů užití. Pod tímto pojmem si můžeme představit vlastnost (schopnost), kterou účastník od systému očekává. Případy užití jsou vždy inicializovány účastníkem a jsou vždy napsány z jeho pohledu. Každý případ užití musí být specifikován. U složitějších případů užití můžeme k jejich specifikaci využít scénářů. Poslední problematikou, na kterou se v rámci tohoto pracovního postupu soustředíme, je alespoň základní identifikace relací mezi jednotlivými účastníky a případy užití.

2. **Analýza.** V rámci tohoto pracovního postupu dochází k upřesňování požadavků a jejich strukturování. Výstupem by měl být analytický model, který se skládá především z diagramu analytických tříd. Je nutné mít na paměti, že termín analytická třída neodpovídá třídě, která bude součástí programového kódu aplikace. Správná analytická třída odpovídá objektu z problémové domény. Identifikuje klíčové vlastnosti objektu a pojmenovává činnosti, které je objekt schopen vykonávat. V rámci analytického modelu tříd tedy nevytváříme přesný návrh, který by sloužil jako podklad pro převod do programového kódu. Není tedy povinné specifikovat všechny atributy a metody atd. Jedná se pouze o problémový rámec, ze kterého budeme v rámci tvorby konečného návrhu vycházet. Dalším výstupem tohoto pracovního postupu by měly být realizace případů užití. Slouží jako ukázky toho, jak by jednotlivé třídy problémové domény měly spolupracovat. K tomu se často využívají jak diagramy spolupráce, tak sekvenční diagramy, které jsou obsahem specifikace jazyka UML.

3. **Návrh.** Na rozdíl od analýzy, která vychází z problémové domény, vychází pracovní postup Návrh z domény řešení. Jeho hlavním úkolem je převod analytického modelu do takové formy, kterou je možné implementovat. Základem je tvorba návrhových podsystémů (dílčí související části výsledného projektu) a návrhových tříd. Ty na rozdíl od svých analytických předchůdců již musí obsahovat všechny implementační detaily, jako jsou datové typy jednotlivých atributů, parametry a návratové metody jednotlivých metod, viditelnost atd. Dalším důležitým výstupem návrhu je hledání rozhranní – společných operací, které některé třídy implementují. Rozhranní je mocný nástroj, v jazyce Java hojně využívaný k podpoře jedné z klíčových vlastností objektového programování – polymorfizmu. Stejně jako nad rozhranními je vhodné se zamyslet nad dědičností a

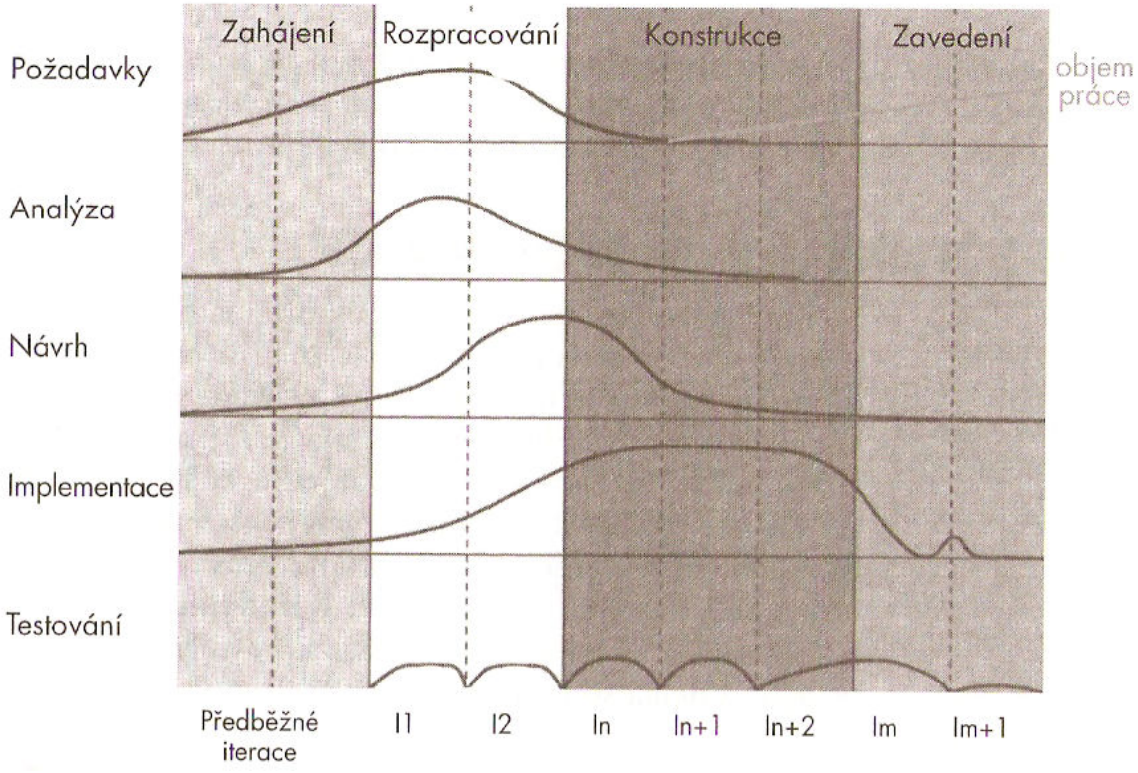
vytvořit správnou hierarchii tříd tak, aby co nejvíce odpovídala modelovanému světu. Dalším možným výstupem tohoto pracovního postupu jsou návrhové realizace případů užití, skládají se z konkrétních (o implementační schopnost rozšířených) návrhových diagramů spolupráce a sekvenčních diagramů. Dalším pojmem, který v souvislosti s návrhem specifikuje jazyk UML jsou diagramy aktivit. I ty je možné využít.

4. **Implementace.** Implementační model je ve skutečnosti pouze pohledem implementace na návrhový model. V jeho rámci můžeme opět využít některých nástrojů, který nabízí specifikace UML. A to konkrétně diagramu komponent, který modeluje závislosti mezi softwarovými komponentami, a diagramu nasazení, který modeluje jednotlivá úskalí fyzického nasazení softwaru, to včetně přehledu hardwarových požadavků, které jsou nezbytné při spouštění systému.
5. **Testování.** Je jedním z nejdůležitějších, ale často podceňovaným, pracovním postupem. Právě v jeho průběhu se rodí hotová aplikace, kterou je možné prezentovat do světa. Testování rozdělujeme na funkční, uživatelské a výkonnostní. Úkolem funkčního testování je ověřit, zda program dělá správně to, co dělat má a zdali vyhovuje specifikaci. Uživatelské testování pak vyžaduje, aby byla alespoň částečně hotové uživatelské rozhraní. Testy by měli provádět budoucí uživatelé, nebo někdo, kdo se jim svými znalostmi a zkušenostmi podobá. Výkonnostní testování pak směřuje k ověření výkonnosti aplikace. [11]

Důraz, který je kladený na jednotlivé pracovní postupy v rámci iterací, závisí na umístění každé konkrétní iterace v kontextu životního cyklu daného projektu. Ten lze rozdělit na čtyři fáze.

- Počáteční fáze (inception) – období plánování,
- rozpracování (elaboration) – období architektury,
- konstrukční fáze (construction) – počátky provozuschopnosti,
- fáze nasazení (transition) – nasazení produktu do uživatelského prostředí. [4]

Jak projekt přechází z jedné fáze do druhé, mění se i objem prací vykonávaných v každém z pěti základních pracovních postupů.



Obr. 6: Objem práce věnovaný jednotlivým fázím životního cyklu aplikace.
Zdroj: ARLOW, J., Neustat, I. UML a unifikovaný proces vývoje aplikací [2]

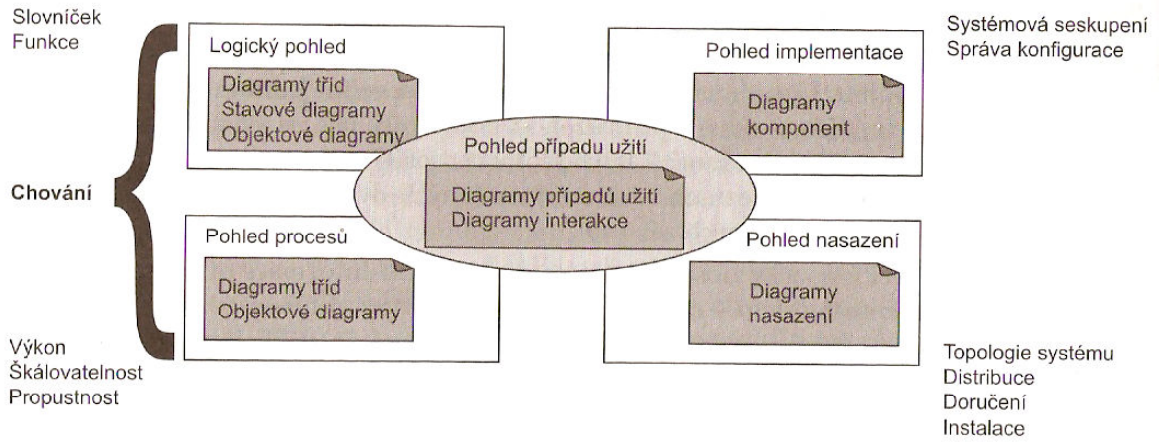
Na horizontální ose jsou na obrázku č. 6 znázorněny jednotlivé fáze, na svislé je umístěno pět základních postupů metodiky UP. Z obrázku vyplývá, že v počáteční fázi je nejvíce času věnováno sestavení požadavků a analýze. Postupně, jak se pohybujeme ve směru životního cyklu aplikace, tyto postupy naopak ustupují do pozadí a ve fázi konstrukce je nakonec důraz zcela zřetelně kladen především na návrh a implementaci. V rámci životní etapy zavedení už zase naopak probíhá už jen implementace a testování. [4]

Aby byla schopna zachytit všechny podstatné aspekty architektury daného systému, využívá metodika UP pohled na architekturu známý jako 4+1 (viz. obrázek č. 7). Ten

specifikuje čtyři základní pohledy (logický, implementační, nasazení a procesní). Každý z těchto pohledů vytváří jistá specifika.

Zatímco logický pohled zachycuje oblast problému jako množinu tříd a objektů, zabývá se statickými vztahy objekty problémové domény, tak pohled procesů se soustřeďuje především na chování systému a omezení plynoucích z případů užití. Jedná se vlastně o procesově orientovanou variantu logického pohledu. Implementační pohled pak zobrazuje soubory a komponenty, které utvářejí hotový kódový základ systému a pohled nasazení zase modeluje fyzické nasazení komponent.

Tyto čtyři pohledy na architekturu aplikace by měly napomoci k celkovému cíli – k naplnění pátého pohledu – pohledu případů užití. Ten předkládá pohled na systém z hlediska jeho funkcionality pro koncového uživatele.



Obr. 7: Architektura metodiky UP.

zdroj: ARLOW, J., Neustat, I. UML a unifikovaný proces vývoje aplikací [2]

3.2 Pracovní postup: Požadavky a jejich specifikace

Tento pracovní postup má tři hlavní cíle. Nalézt hranice systému, účastníky a případy užití.

Vyjděme z představy, jak by měl systém obchodního domu vypadat. Měl by umožňovat existenci různých oddělení (samostatně vyhlížejících elektronických obchodů). Měl by ale zpřístupňovat jednotné a pro uživatele průhledné administrátorské rozhraní. Začněme tedy nejprve například hledáním účastníků.

Při hledání účastníků vycházíme z toho, kdo bude systém obchodního domu využívat. Na jedné straně jsou to nepochybně zákazníci, kteří za jeho pomoci budou nakupovat. Nesmíme ale zapomínat ani na uživatele, kteří jsou pro běh samotného systému nepostradatelní (správci atd.).

Po úvaze tedy v systému můžeme rozlišovat čtveřici účastníků.

1. Zákazník – využívá veřejných služeb obchodního domu.
2. Správce obchodního domu – má za úkol spravovat jednotlivá oddělení.
3. Správce oddělení – má za úkol spravovat jednotlivé úseky (knihkupectví, elektor) obchodního domu.
4. Manažér objednávek – má za úkol spravovat systém z pohledu správy objednávek. Je odpovědný za jejich plnění.

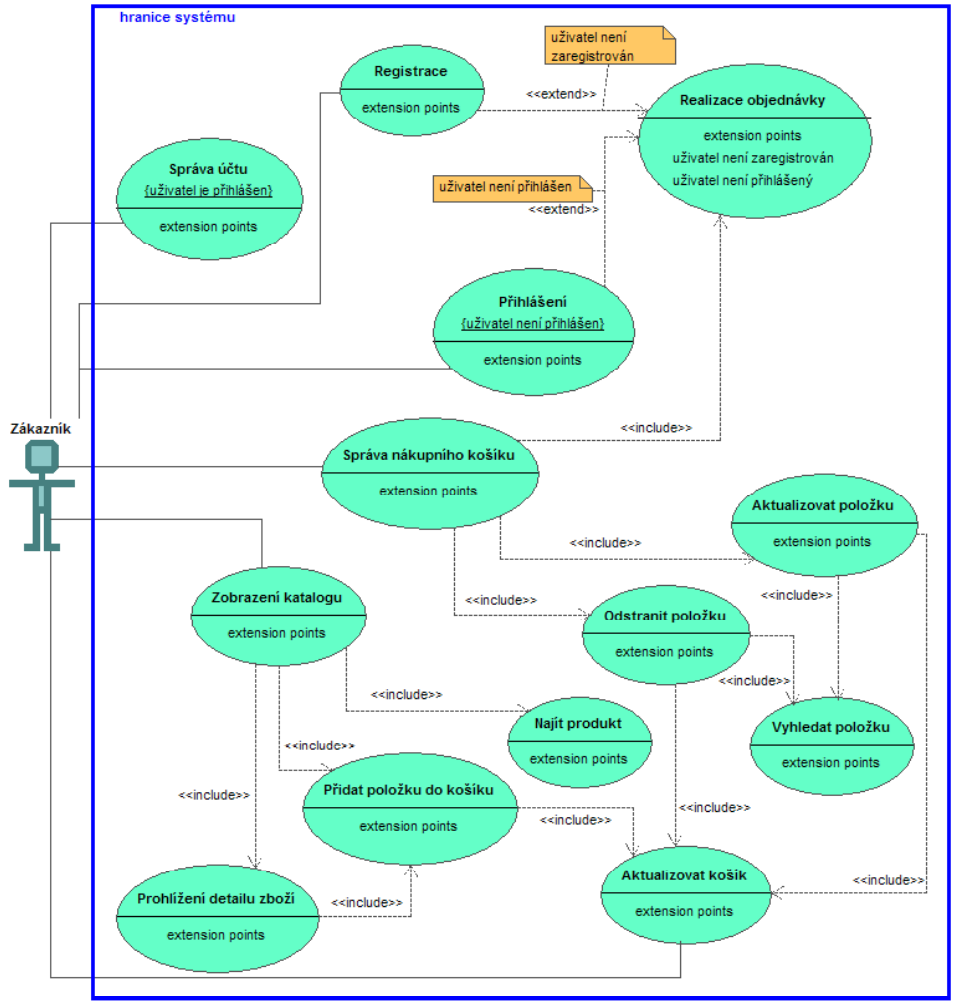
Tuto čtveřici by šlo samozřejmě ještě rozšířit. Pro zachování přehlednosti této diplomové práce budou ale nadále uvažováni pouze tyto čtyři. Pro demonstraci i základní funkcionalitu zcela dostačují.

Každý z těchto účastníků bude mít i jiná práva, která mu v systému obchodního domu budou náležet. Zákazník by například neměl v žádném případě získat přístup do administrace systému. I ostatní uživatelé by měli mít přístup do administrace omezený, v

závislosti na jejich příslušnosti k jednotlivým rolím. Aby například nemohlo dojít k situaci, kdy manažér objednávek bude manipulovat s katalogem zboží, či dokonce zakládat či rušit celá oddělení v systému vedená. Každý z účastníků by tedy měl využívat systém jen k tomu, k čemu je svým posláním předurčen.

A tak se dostávám k dalším bodům tohoto pracovního postupu, tj. vyhledávání případů užití a hranic systému. Co bude tedy konkrétní náplní poslání, která každému z účastníků plynou? K prezentaci využiji znalostí o modelech případů užití.

Začněme nejprve od účastníka „masového“ - tj. zákazníka obchodního domu. Na následujícím obrázku můžete vidět shrnující model případů užití účastníka reprezentujícího zákazníka.



Obr. 8: Diagram případů užití zákazníka.

Zdroj: vlastní.

Jednotlivé případy užití je ale třeba specifikovat, aby byl model úplný.

3.2.1 Případ užití: Přihlášení

Tab. 3: Specifikace případu užití přihlášení.

Účastníci: Zákazník *Manažér objednávek* *Správce oddělen* *Správce obchodního domu*

Vstupní podmínky: Uživatel není přihlášen.
Tok událostí: 1. Případ užití začíná, klikne-li účastník na odkaz Přihlásit se. 2. Uživatel zadá uživatelské jméno a heslo. 3. Systém zkontroluje správnost vložených údajů. 4. KDYŽ kontrola proběhne úspěšně, je účastník: ● přesměrován na svou domovskou uživatelskou stránku. 5. DOKUD kontrola vykazuje chybu, je účastník: ● vyzván k opětovnému zadání údajů, ● schopen vygenerovat nové heslo a obdržet ho emailem, ● je vyzván k registraci.
Následné podmínky: Stav uživatele je aktualizován a jsou mu nově zobrazeny všechny oblasti, které jsou přístupné jen přihlášeným uživatelům.

Zdroj: vlastní.

Pro úplnost bych se chtěla zastavit o formě bloku Účastníci ve specifikaci případu užití Přihlášení. Ačkoli je zatím mou snahou vyhledávat případy užití účastníka zákazník, je dobré mít na paměti, že i ostatní účastníci budou v systému využívat přihlašovacího mechanizmu. Stejně tak jako budou schopni pod svými údaji nakupovat, nakládat se svými osobními údaji, prohlížet katalog atd. V rámci přehlednosti modelu jsem se rozhodla neuvádět ty případy užití, které budou totožné, do diagramů následujících účastníků. Ve specifikaci budou ale uváděny vždy.

3.2.2 Případ užití: Registrace

Tab. 4: Specifikace případu užití registrace.

Účastníci: Zákazník
Tok událostí: 1. Účastník zadá povel [Registrovat se] 2. Vyplní požadované uživatelské jméno. 3. KDYŽ uživatelské jméno v systému existuje, je vyzván pro opětovné zadání jiného. 4. Vyplní 2x heslo (pro ověření). 5. KDYŽ hesla nesouhlasí, je vyzván k nápravě.

6. Vyplní ostatní povinné údaje – jméno, příjmení, email, adresu, telefon.
Následné podmínky: Uživateli je odeslán email s registračními údaji a je přesměrován na stránku s přihlašovacím formulářem.

Zdroj: vlastní.

Na rozdíl od výše zmíněného případu užití Přihlášení si všimněte, že registrace není sdíleným případem s „nadřazenými“ rolemi. Je tomu tak proto, že „zaměstnance“ má právo vytvářet jen nadřazená entita – tj. správce oddělení, či správce obchodního domu. Není možné je tedy vytvářet pouhou registrací.

3.2.3 Případ užití: Správa nákupního košíku

Tab. 5: Specifikace případu užití Správa nákupního košíku.

Účastníci: Zákazník *Manažér objednávek* *Správce oddělení* *Správce obchodního domu*
Vstupní podmínky: 1. Nákupní košík není prázdný. 2. Obsah nákupního košíku je viditelný – tzn. uživatel například kliknul na obrázek reprezentující nákupní košík.
Tok událostí: 1. Případ užití začíná, klikne-li uživatel na nějakou položku v košíku. 2. KDYŽ uživatel zadá povel „smazat položku“, systém odstraní položku z košíku. 3. KDYŽ uživatel zadá nové množství, ● systém aktualizuje množství, ● systém přepočítá výslednou cenu.
Následné podmínky: Obsah košíku byl aktualizován.

Zdroj: vlastní.

I tento případ užití je vhodné vést jako sdílený s „nadřazenými“ účastníky. A proč to může být výhodné? Představte si třeba situaci, kdy budete chtít nabízet svým zaměstnancům

zaměstnaneckou slevu. V případě, že ostatním uživatelům (manažérům objednávek, správcům oddělení) umožníte nakupovat, umožníte tím také jejich snadnou identifikaci do role Vašeho zaměstnance a přeměnit to tak v různé výhody.

3.2.4 Případ užití: Spravování účtu

Tab. 6: Specifikace případu užití Spravování účtu.

Účastníci: Zákazník *Manažér objednávek* *Správce oddělení* *Správce obchodního domu*
Vstupní podmínky: Účastník je přihlášen do systému.
Tok událostí: 1. Případ užití začíná, vybere-li účastník volbu [Osobní údaje]. 2. Uživatel zkontroluje, popř. změní vybrané položky ve formuláři. 3. Uživatel potvrdí změny pomocí potvrzovacího tlačítka.
Následné podmínky: Uživatelovy osobní údaje jsou aktualizovány.

Zdroj: vlastní.

I tento případ užití můžeme považovat za sdílený. Nicméně je vhodné poznamenat, že zákazníci a zaměstnanci mohou mít v rámci editace osobních dat dostupné odlišné verze editačních formulářů.

3.2.5 Případ užití: Zobrazení katalogu zboží

Tab. 7: Specifikace případu užití Zobrazení katalogu zboží.

Účastníci: Zákazník *Manažér objednávek* *Správce obchodního domu*

Správce oddělení
Tok událostí: 1. Případ užití začíná, když: ● uživatel vstoupí na domovskou stránku obchodního domu, ● uživatel vstoupí na domovskou stránku oddělení, ● když klikne na některou z kategorií v rámci oddělení. 2. KDYŽ uživatel vyplní specifický filtr, ● katalog se znovu načte, ● zobrazí se jen odpovídající produkty. 3. KDYŽ klikne na odkaz [Přidat do košíku], ● zboží se přidá do košíku, ● objeví se ikona pro vyvolání správy nákupního košíku. 4. KDYŽ účastník klikne na položku v katalogu, ● je přesměrován na její detail s doplňujícími informacemi. 5. KDYŽ účastník zadá povel [Setřídit podle...] ● katalog se znovu načte, ● produkty se zobrazí v odpovídajícím pořadí.

Zdroj: vlastní.

Všimněme si, že na prohlížení katalogu zboží nemusí být uživatel přihlášený do systému. To je jistě žádoucí, nechceme přeci uživatele zbytečně „otravovat“ a vyžadovat po něm přihlášení vždy, když jen zavítá na domovskou stránku našeho obchodního domu. Autorizace je však nepochybně nutná v případě úspěšného dokončení objednávky.

3.2.6 Případ užití: Realizace objednávky

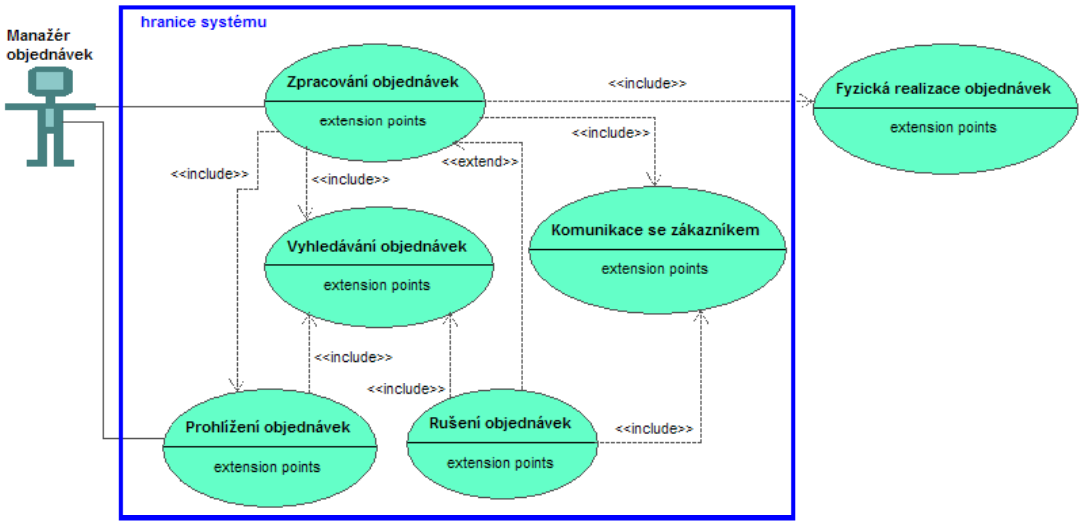
Tab. 8: Specifikace případu užití Realizace objednávky

Účastníci: Zákazník *Manažér objednávek* *Správce oddělení* *Správce obchodního domu*
Vstupní podmínky: Nákupní košík není prázdný.
Tok událostí: 1. Případ užití začíná ve chvíli, kdy účastník zadá povel [Dokončit objednávku]. 2. KDYŽ účastník není přihlášený, ● je vyzván k zadání přihlašovacích údajů,

• přesměrován zpět na první stránku s dokončením objednávky – správu nákupního košíku. 3. KDYŽ účastník není registrovaný, • je vyzván k naplnění případu užití Registrace, • automaticky přihlášen do systému, • a přesměrován zpět na první stránku s dokončením objednávky – správu nákupního košíku. 4. Účastník může spravovat nákupní košík (viz. Případ užití Správa nákupního košíku). 5. Účastník potvrdí svou objednávku. 6. Účastník má možnost editovat své osobní údaje vztahující se k objednávce (výchozí jsou ty uvedené v osobních datech). 7. Účastník odešle objednávku.
Následné podmínky: • Objednávka se založí v systému s odpovídajícím stavem „K VYŘÍZENÍ“. • Uživateli je odeslán automatický email se shrnujícími informacemi. • Správci oddělení je zasláno upozornění o přijetí nové objednávky a ten na jeho základě deleguje odpovědného manažéra objednávky.

Zdroj: vlastní.

Pojďme se nyní zaměřit na manažéra objednávek a jeho požadavky vůči systému. Shrnuje je následující diagram případů užití.



Obr. 9: Diagram případů užití manažéra objednávek

Zdroj: vlastní.

3.2.7 Případ užití: Zpracování objednávek

Tab. 9: Specifikace případu užití Zpracování objednávek.

Účastníci: Manažér objednávek
Vstupní podmínky: Uživatel je přihlášen do systému. Seznam objednávek, či upozornění na nově příchozí objednávku, jsou viditelné.
Tok událostí: 1. Případ užití začíná ve chvíli, kdy manažér objednávek přijme přidělenou objednávku (změní její stav na VYŘIZUJE SE). 2. KDYŽ je objednávka v pořádku, ● zablokuje zboží vedené v systému pro její účely, ● **za hranicemi systému** provede všechny akce související s plněním objednávky, ● po odeslání objednávky změní její stav v systému na ODESLÁNO. 3. KDYŽ není objednávka v pořádku, ● pomocí systému informuje zákazníka o problémech, ● po dohodě objednávku zruší, či ji provede ve změněné formě.
Výstupní podmínky: Uživateli je systémem odeslán mail s informacemi o tom, kdy byla objednávka expedována a kdy může očekávat její doručení.

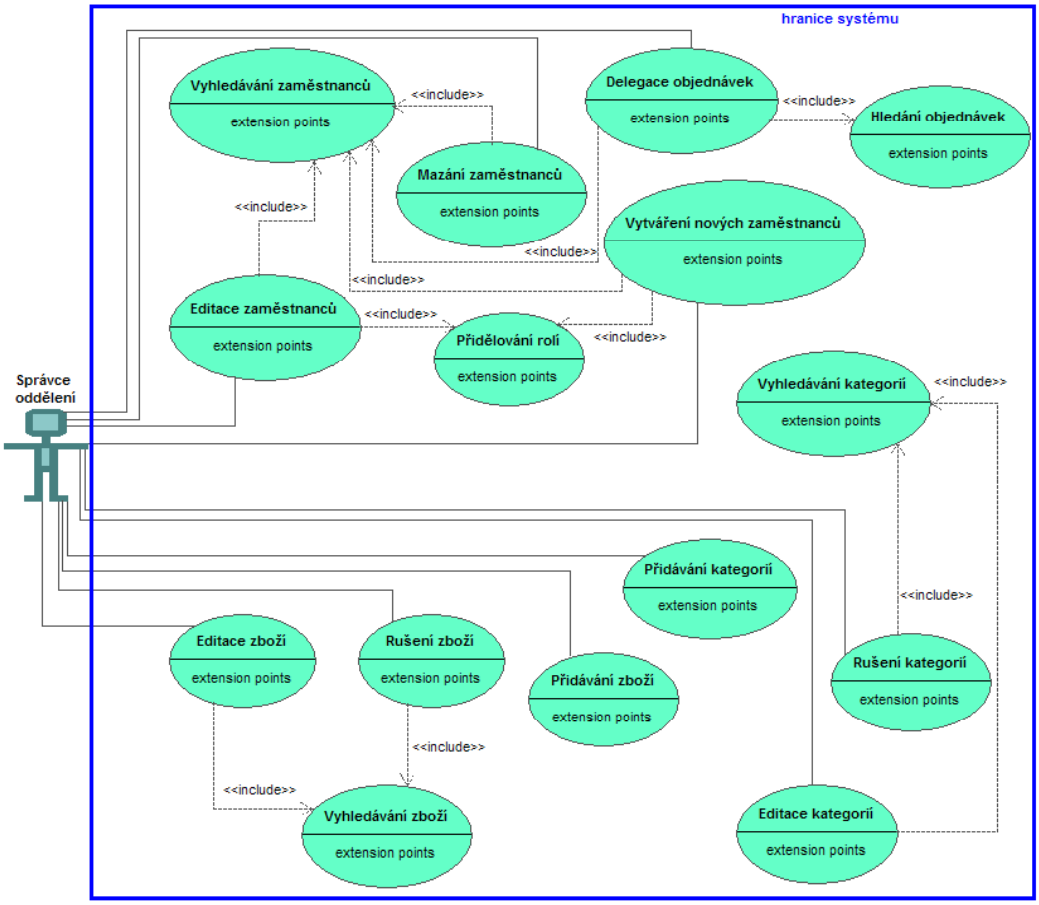
Zdroj: vlastní.

V tomto případě užití se poprvé setkáváme s důležitostí pojmu hranice systému, a to v případu užití nazvaném Fyzická realizace objednávek. Pod tímto pojmem si můžeme představit například balení, expedici, jednání s doručovacími společnostmi. Tyto informace nejsou pro systém nijak podstatné, není tedy nutné je do systému zavádět. Proto je tento případ užití umístěn až za hranicemi systému a dále nebude rozebírán.

Stejně tak bychom mohli specifikovat dvojici případů užití Komunikace se zákazníkem. V diagramu jsem tento případ užití umístila dovnitř systému. K tomu, aby byla tato skutečnost správně pochopena, je nutné ujasnit si, co se pod tímto pojmem skrývá. Pod tímto případem užití je myšlena komunikace v rámci systému – tj. veškeré automatické emaily, které zákazník obdržel, stejně tak jako komunikace mezi manažérem objednávky a zákazníkem (např. při problémech s vyřízením objednávky). Je totiž vhodné, aby se o této komunikaci vedly v aplikaci záznamy (ať už pro archivaci, či pro kontrolu vedenou nadřízenými). Vedle této komunikace může ale existovat i komunikace vedená mimo

systém – například telefonické ověřování objednávek, která se v případě nutnosti musí do systému zadat ručně.

Chtěla bych se také pozastavit nad případem užití Rušení objednávek. Tento případ je v diagramu zachycen jako rozšiřující (extend) právě případu užití Zpracování objednávek. Tato skutečnost souvisí s bodem tři v toku událostí. Rušení objednávek je totiž opravdu možné brát jako speciální druh zpracování objednávky. A to v případě, kdy dojde k nějakým problémům. Například objednané zboží není náhle dostupné atd. Proto tedy ono klíčové slovo extend. Pojďme se nyní zaměřit na dalšího účastníka – správce oddělení. Jeho diagram případů užití vidíme na následujícím obrázku.



Obr. 10: Diagram případů užití správce oddělení

Zdroj: vlastní.

3.2.8 Případ užití: Delegace objednávek

Tab. 10: Specifikace případu užití Delegace objednávek.

Účastníci: Správce oddělení v komunikaci s manažérem objednávek
Vstupní podmínky: Správce oddělení musí být přihlášený do systému. List objednávek k vyřízení musí být viditelný.
Tok událostí: 1. Případ užití začíná ve chvíli, kdy správce oddělení klikne na položku v seznamu objednávek k vyřízení. 2. Správce oddělení vybere ze seznamu manažéra objednávky, kterému bude objednávka předána na odpovědnost. 3. Správce oddělení potvrdí svůj výběr. 4. Vybranému manažéru objednávky přijde upozornění na nový úkol.
Následné podmínky: List nových nepřidělených objednávek se znovu načte.

Zdroj: vlastní.

Další skupina případů užití souvisí s pravomocemi správce oddělení manipulovat se zaměstnanci. Správce oddělení je totiž pravomocný zakládat do systému nové manažéry objednávek a nové správce oddělení (a tak sdílet s jiným uživatelem své pravomoci). Naopak je dobrým zvykem, aby uživatel systému nemohl zakládat uživatele s většími pravomocemi – proto by správce oddělení neměl být schopen zakládat správce obchodního domu. Pojďme podrobněji rozebrat případ užití Vytváření nových zaměstnanců, který v sobě obsahuje právě i přidělování rolí.

3.2.9 Případ užití: Vytváření nových zaměstnanců

Tab. 11: Specifikace případu užití Vytváření nových zaměstnanců.

Účastníci: Správce oddělení Správce obchodního domu

Vstupní podmínky: Účastník je přihlášen do systému.
Tok událostí: 1. Případ užití začíná ve chvíli, kdy účastník zadá povel [Přidat nového zaměstnance]. 2. Účastník vyplní povinné údaje nového zaměstnance (jméno, příjmení, email, přihlašovací jméno) a potvrdí formulář. 3. KDYŽ je uživatel s přihlašovacím jménem již v systému evidován: ● je účastník vyzýván k zadání nového přihlašovacího jména. 4. Účastník vybere roli nového uživatele. 5. Účastník potvrdí nového uživatele do systému. 6. Systém odešle na email zadaný ve formuláři údaje nutné pro vstup do systému – včetně automaticky vygenerovaného hesla.
Následné podmínky: Nový uživatel se může přihlásit do systému a po přihlášení bude správně (dle definované role) autorizován a schopen plnit své úkoly.

Zdroj: vlastní.

Další případy užití souvisí s pravomocemi ohledně spravování katalogu zboží a struktury katalogu vůbec. Zboží v každém elektronickém obchodě je totiž členěno do kategorií, aby bylo pro zákazníky lehce dohledatelné. Právě správu těchto kategorií jsem se po úvaze rozhodla přidělit správcům oddělení (a ve skutečnosti i správcům obchodních domů).

Stejně tak mají správci oddělení na starosti přidávat do katalogu nová zboží, či rušit naopak taková, která nejsou již více dostupná. Protože přidávání (resp. rušení a editace) kategorií a zboží (stejně tak jako přidávání oddělení, které bude v pravomocích správce obchodního domu) jsou případy užití velmi podobné a liší se pouze v předmětu toho daného případu užití, proberu zde podrobněji jen případy související s kategoriemi. Ostatní si od nich lze lehce odvodit.

3.2.10 Případ užití: Přidávání kategorií

Tab. 12: Specifikace případu užití Přidávání kategorií.

Účastníci: Správce oddělení

Správce obchodního domu
Vstupní podmínky: Účastník je přihlášen do systému.
Tok událostí: 1. Případ užití začíná, zadá-li účastník povel [Přidat kategorii]. 2. Účastník vyplní náležitosti týkající nové kategorie – jméno atd. 3. Účastník potvrdí novou kategorii. 4. Nová kategorie je zařazena do systému v rámci DEMO módu. 5. Po potvrzení DEMO módu je kategorie (a v rámci DEMO módu do ní převedené zboží) převedena do ostrého provozu.
Následné události: V oddělení je pro návštěvníky dostupná nová kategorie.

Zdroj: vlastní.

Pojďme se chvíli pozastavit nad termínem DEMO mód. Jistě totiž není vhodné, aby se při editaci struktury oddělení, či celého obchodního domu, měnil obchod zákazníkovi pod rukama. Jistě nechceme, aby uživatel viděl prázdné kategorie, či celá prázdná oddělení obchodního domu. Ani kamenné obchody se neotvírají v době, kdy ještě nejsou pro oko zákazníka perfektně připravené. V elektronickém obchodování platí stejná pravidla. Proto tedy DEMO mód.

Správci oddělení vidí v rámci administrace systému jeho DEMO podobu, která odpovídá ostré verzi s DEMO změnami. Zatímco zákazník vidí pouze ostrou verzi. Až je správce se změnami spokojený – měl by být schopen naráz převést všechny změny do ostrého provozu. A až se tak stane, měly by být zákazníkovi přístupné i nové kategorie (či zboží, oddělení).

3.2.11 Případ užití: Mazání a editace kategorií

Tab. 13: Specifikace případu užití Mazání a editace kategorií.

Účastníci: Správce oddělení Správce obchodního domu

Vstupní podmínky: Účastník je přihlášen do systému. List kategorií je viditelný.
Tok událostí: 1. Případ užití začíná ve chvíli, kdy účastník klikne na kategorii v seznamu. 2. KDYŽ účastník zadá povel smazat kategorii: ● kategorie je smazána ze systému v rámci DEMO módu, ● po potvrzení demo módu je kategorie odstraněna i z ostré verze systému. 3. KDYŽ účastník zadá povel k editaci kategorie: ● uloží se aktualizovaná data patřící kategorii do DEMO módu, ● po potvrzení demo módu je kategorie aktualizovaná i v ostrém provozu.
Následné podmínky: Systém je pro zákazníky dostupný v ostré verzi.

Zdroj: vlastní.

3.3 Pracovní postup: Analýza

Dalším krokem v objektovém návrhu systému obchodního domu je tvorba analytického modelu. S tím souvisí hledání analytických tříd. Tyto třídy nemusí být přesně definovány co se programátorských náležitostí týče (včetně kompletně definovaných atributů a metod), mají hlavně vyjadřovat podstatu problémové domény analyzovaného systému.

V případě obchodního domu vyjdeme z požadavků definovaných v diagramech případů užití. Vyjděme nejprve z definice účastníků a hlavně pak z předpokladu, že s každým účastníkem bude možné nakládat jako se zákazníkem. Tzn. že uživatel vedený jako správce oddělení bude mít volnou ruku ve využívání systému pod svým uživatelským jménem a heslem, avšak pouze pro účely nakupování.

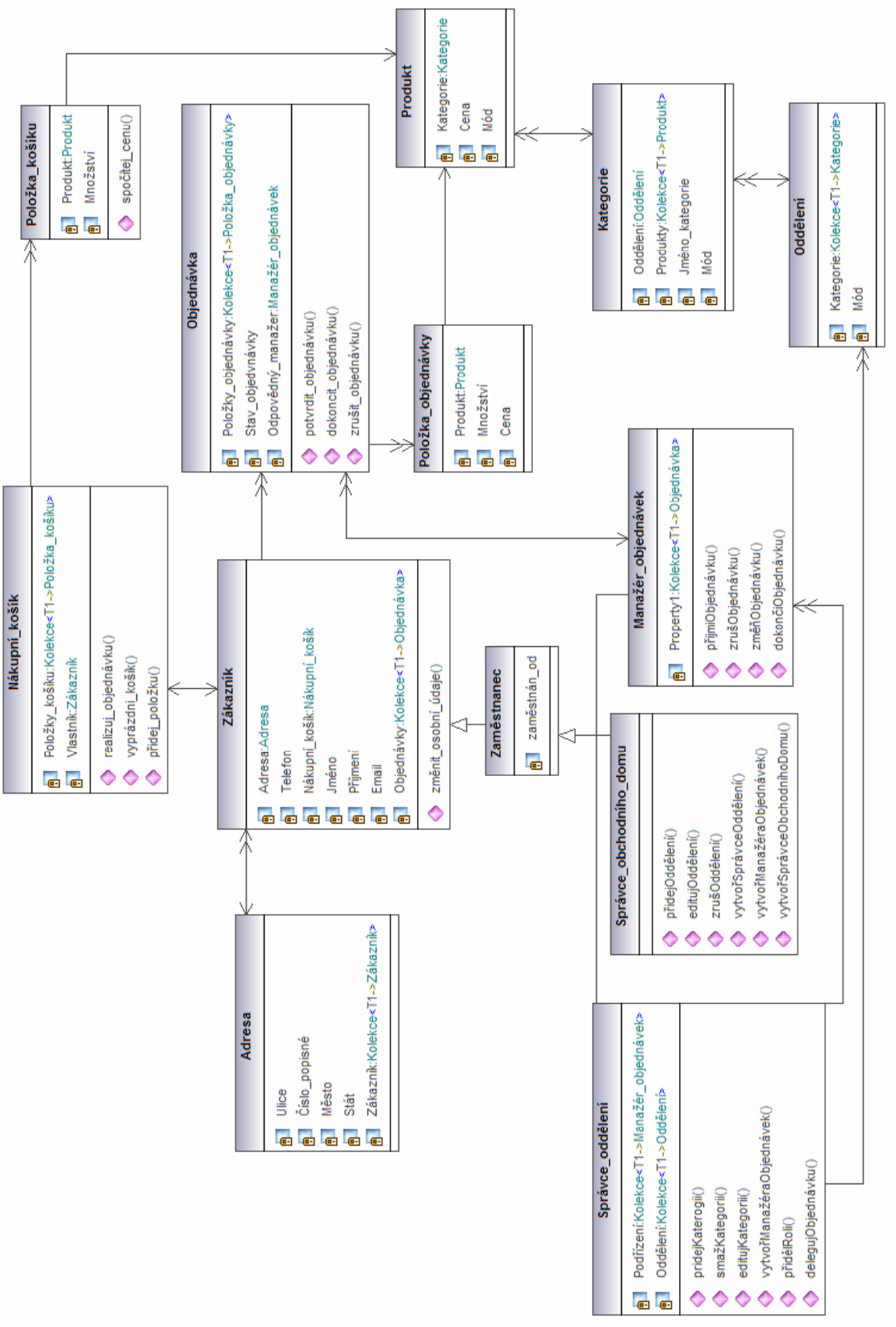
Z tohoto předpokladu tedy nejprve odvodíme analytickou třídu Zákazník. Rozšířením této třídy se dostaneme ke třídě Zaměstnanec, která rozšiřuje vlastnosti rodiče Zákazník o atributy směrodatné pouze pro zaměstnance. Z této třídy můžeme následně odvodit i její konkrétní potomky – správce oddělení, manažéra objednávek a správce obchodního domu. Každá z těchto specifických tříd zastupuje určitou roli, kterou uživatelé nabývají a jsou

díky ní schopni provádět specifické operace. Správce oddělení je například schopen vytvářet nové kategorie, zatímco manažér objednávek nikoli.

Další analytické třídy odvodíme z nákupního procesu. Tento proces samozřejmě vyžaduje přítomnost tříd reprezentující nákupní košík a jednotlivé položky v něm umístěné. Každá z položek pak odkazuje na nějaký konkrétní produkt z katalogu zboží. Když se uživatel rozhodne dokončit nákup – stává se z obsahu nákupního košíku obsah objednávky (proto analytická třída Objednávka). Ta si udržuje vazbu na uživatele – obsah každé objednávky je totiž vhodné archivovat, aby se zákazník mohl zpětně podívat na seznam svých realizovaných objednávek.

Stejně tak v analytickém modelu tříd sledujeme závislosti mezi produkty a jednotlivými kategoriemi (relace mezi analytickými třídami Produkt a Kategorie) a mezi kategoriemi a odděleními (relace mezi třídami Kategorie a Oddělení). Platí tu jednoduché pravidlo – každá kategorie bude obsahovat několik produktů, zatímco každý produkt bude náležet právě jedné kategorii. Stejně tak každé oddělení může mít více kategorií, které pomáhají zákazníkovi orientovat se v systému. Ale každá taková kategorie musí náležet právě jednomu oddělení. Mezi všemi těmito třídami tedy můžeme identifikovat relace 1:n.

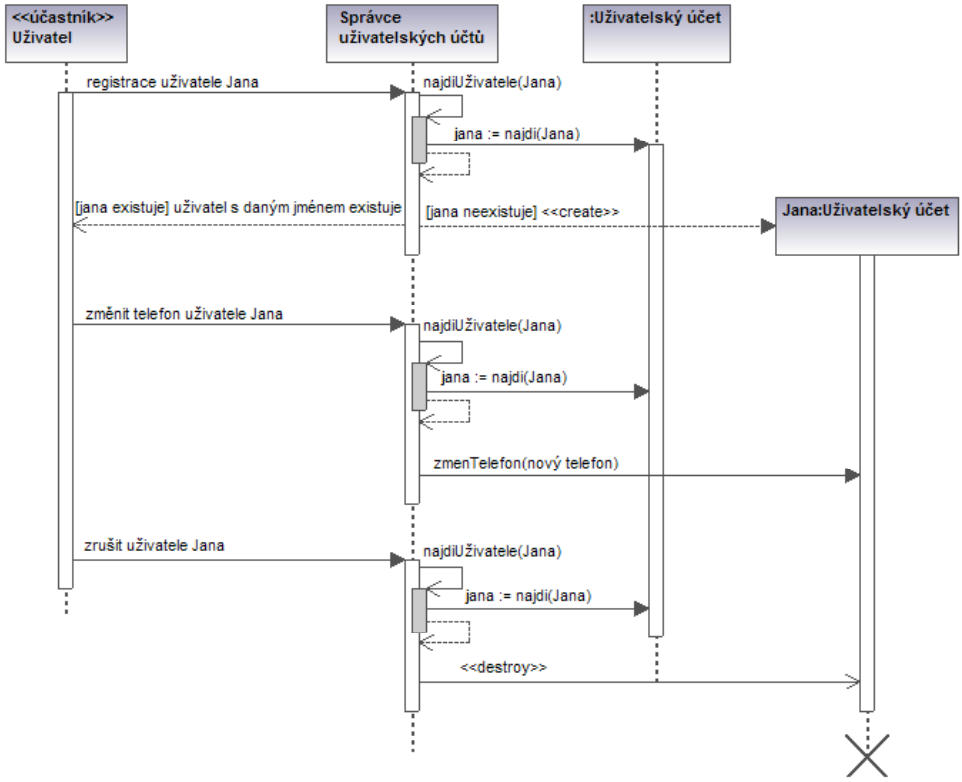
Stejně tak relace 1:n patří mezi dvojici tříd Zákazník a Adresa. V jednom elektronickém obchodě například mohou nakupovat všichni členové jedné rodiny – s oddělenými uživatelskými účty, ale společnou adresou pro doručení zboží. Stejně tak mohou se může stát, že více zaměstnanců vedených v systému bude sdílet jednu společnou adresu. Vztah mezi třídami Zákazník a Adresa v analytickém diagramu tříd tedy také mapuje relace 1:n.



Obr. 11: Analytický diagram tříd systému obchodního domu

Zdroj: vlastní.

Pojďme se ještě podívat na analytický model z dynamického pohledu. K tomu využiji sekvenční diagramy, na kterých se pokusím zachytit vybrané funkce systému. První z nich znázorňuje problém registrace nového uživatele, změnu osobních dat a následné zrušení celého účtu.

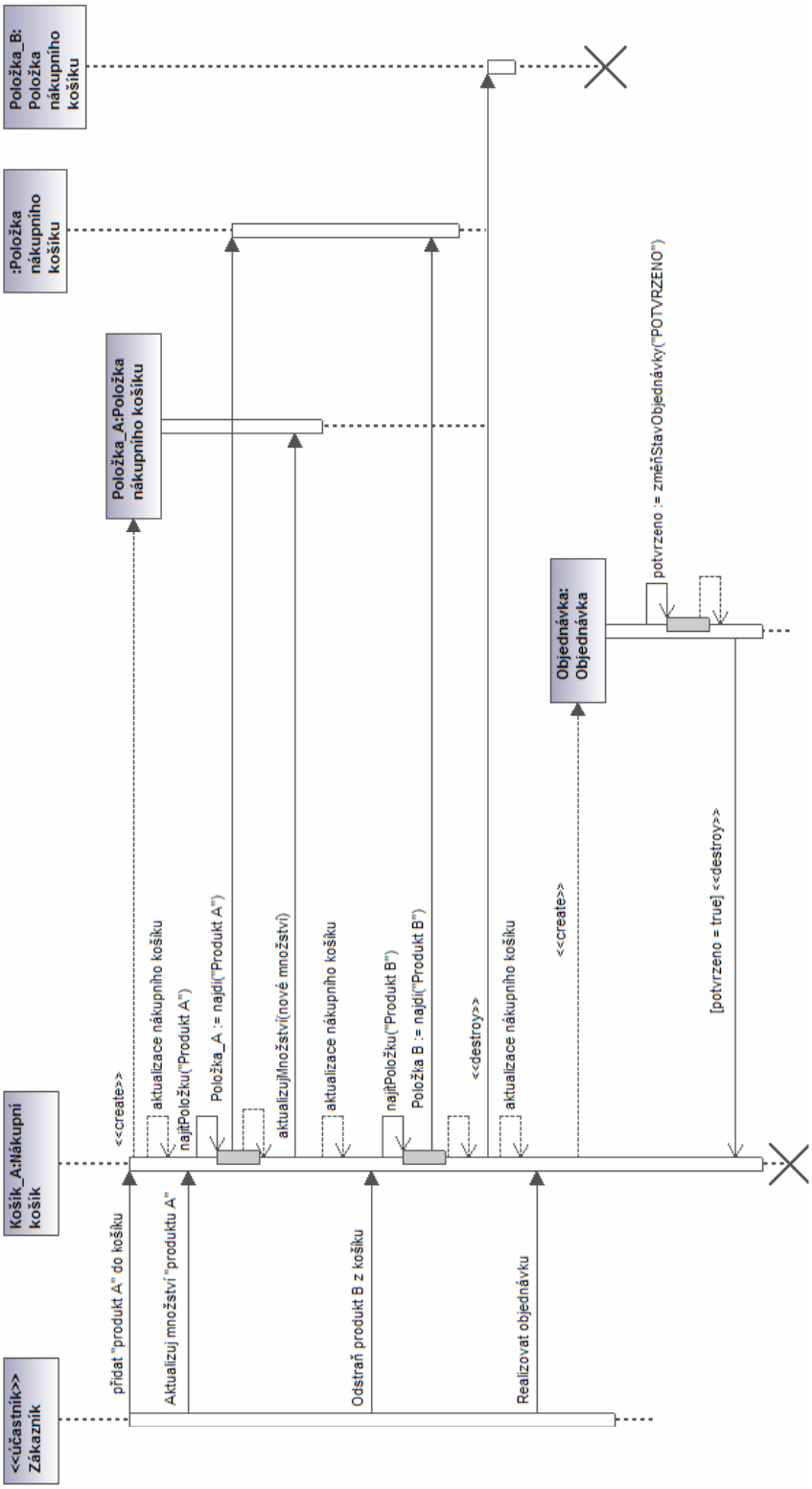


Obr. 12: Sekvenční diagram zachycující aktivity nad uživatelským účtem

Zdroj: vlastní.

Na obrázku 12 je jasně vidět, jak by měl celý proces probíhat. Nejprve uživatel zadá povel k nové registraci. Jako jeden z prvních úkonů se provede kontrola, zdali uživatel s daným uživatelským jménem v systému doposud neexistuje. Pokud je tato kontrola úspěšná – nový uživatelský účet je založen a je automaticky prolinkován s rolí zákazník. Na stejném obrázku můžeme také vidět postup při editaci osobních údajů, či rušení uživatelského účtu. V souladu s případy užití je vidět, že všechny tyto operace (editace, rušení) vyžadují funkci na vyhledávání daného uživatele v systému.

Na dalším sekvenčním diagramu na obrázku 13 je zase zobrazen proces nakupování.



Obr. 13: Sekvenční diagram zachycující proces nakupování.

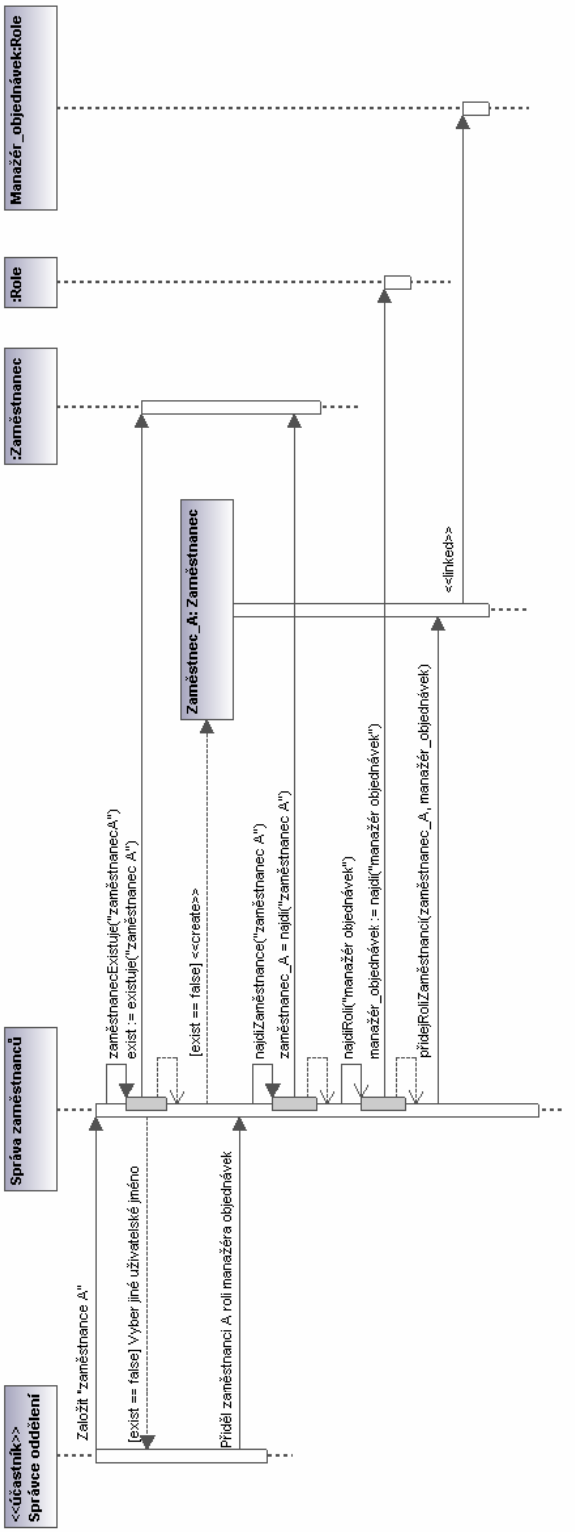
Zdroj: vlastní.

V tomto diagramu je zobrazen postup od přidání zboží do nákupního košíku, přes aktualizaci jeho množství. Po každé manipulaci s nákupním košíkem je nutné stav nákupního košíku obnovit (viz následná podmínka případu užití Správa nákupního košíku).

Dalším důležitým bodem na tomto diagramu je realizace objednávky. Ve chvíli, kdy účastník zadá povel, je obsah nákupního košíku převeden do objednávky a poté, co zákazník objednávku potvrdí, je nákup považován za dokončený. Ve chvíli, kdy se tak stane může také zaniknout instance nákupního košíku.

Stejně jako v případě registrace, změny osobních dat a rušení uživatelského účtu – i zde můžeme vidět, že operace sloužící pro správu nákupního košíku využívají služeb vyhledávání v něm (stejně tak služby aktualizace nákupního košíku).

Další důležitou oblastí, jejíž požadované fungování je v rámci analýzy vhodné nastínit, je proces zakládání nových zaměstnanců do systému. Tento proces poodhaluje následující sekvenční diagram na obrázku 14.



Obr. 14: Sekvenční diagram zachycující založení nového manažéra objednávek

Zdroj: vlastní.

Správce oddělení nejprve vydá povel k založení nového zaměstnance s konkrétním

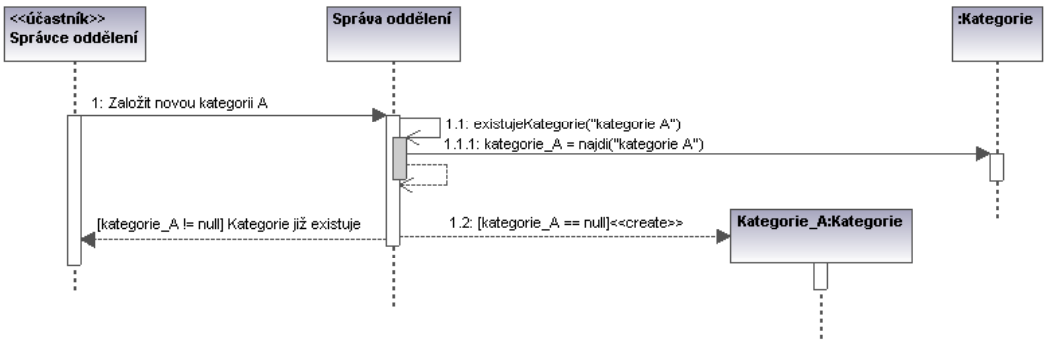
identifikátorem – uživatelským jménem. V rámci tohoto procesu musí samozřejmě dojít ke kontrole, zdali uživatel s požadovaným jménem již v systému nefiguruje. Pokud tomu tak není, zaměstnanci se v systému založí nový účet.

V dalším kroku pak správce oddělení musí novému zaměstnanci přidělit správné pravomoci. Učiní tak díky výběru správné přístupové role – v případě diagramu na obrázku se jedná o roli Manažéra objednávek. Poté, co je tato role v systému nalezena, může dojít k prolinkování vazeb mezi jí a nově založeným uživatelem.

Od chvíle, kdy je proces dokončen, se může nový uživatel do systému přihlásit s jistotou, že bude správně autorizován a bude tak schopen plnit své úkoly.

Poslední problém, který si v rámci analýzy zaslouží pozornost, je zakládání nových kategorií v rámci oddělení. Tato problematika, stejně jako u příbuzných případů užití, má mnoho společného například s problémem zakládání nových oddělení správcem obchodního domu, či přidáváním zboží do katalogu jednotlivých oddělení. Bude tedy podrobněji probírána pouze jednou – a to právě na příkladu zakládání kategorií.

Postup zobrazuje následující sekvenční diagram na obrázku 15.



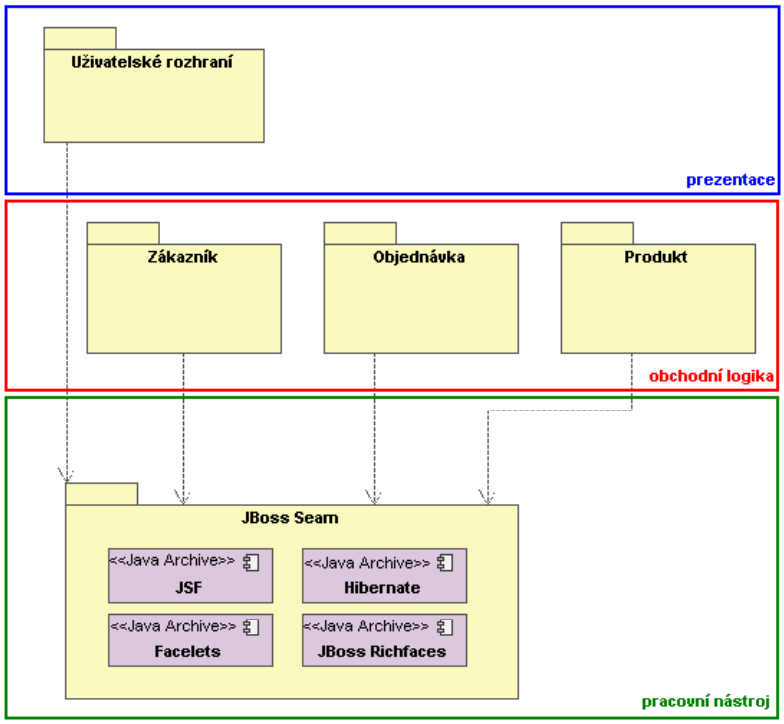
Obr. 15: Sekvenční diagram zachycující založení nové kategorie.

Zdroj: vlastní.

Správce oddělení nejprve vydá povel k založení nové kategorie a přiřadí k požadavku jméno nově vznikající kategorie. Pokud kategorie v rámci systému ještě neexistuje, může být v dalším kroku vytvořena. Pokud naopak existuje, správce oddělení by měl obdržet varovnou hlášku a měl by být požádán k zadání nového (v systému ještě neexistujícího) názvu.

3.4 Pracovní postup: Návrh

Systém obchodního domu je komplexní a poměrně rozsáhlý systém. V rámci pracovního postupu Návrh se budu soustředit na vybrané oblasti. Všechny tyto oblasti budou přímo, či nepřímo souviset integračním frameworkem JBoss Seam. Opodstatnění tohoto kroku je jasně vidět na obrázku 16.



Obr. 16: Část systému obchodního domu prezentovaná ve vrstvách.
Zdroj: vlastní.

Aplikační framework Jboss Seam je frameworkem integračním. Který kromě toho, že poskytuje některé své funkce (konverzační kontext, akce spojené se stránkou atd.),

především koordinuje spolupráci již existujících frameworků – jako jsou JSF, Hibernate (pro přístup k databázím) a další. Všechny tyto samostatné projekty jsou součástí distribuce Seamu a není nutné je tedy nasazovat každý zvlášť.

Z obrázku 16 je také jasně vidět, že jak tvorba prezentační vrstvy s využitím technologií Facelets, JSF a bohaté komponentové knihovny Jboss Richfaces, tak i tvorba aplikační logiky, bude v úzké spolupráci s frameworkem Seam.

Cílem pracovního postupu Návrh je vytvořit návrhový model systému právě za předpokladu využití tohoto integračního frameworku. Protože je ale systém obchodního domu rozsáhlý a kompletní návrhový model by sahal za rámec této diplomové práce, vytyčím si hlavní oblasti, které budou v této části vyzdvihnuty. Budou to následující.

1. Podpora CRUD operací v rámci frameworku Jboss Seam.
2. Bezpečnost, autentizace a autorizace uživatelů a spolupráce s frameworkem Jboss Seam v této oblasti.
3. Zasazení jednotlivých částí systému do kontextů frameworku Seam a vymezení hlavních úrovní konverzací.
4. Nastínění problematiky návrhu prezentační vrstvy za pomocí komponentové knihovny JBoss Richfaces.

3.4.1 CRUD operace

Systém obchodního domu je, stejně jako naprostá většina e-business aplikací, založen právě na operacích Create, Read, Update a Delete – na manipulaci s daty. Ať už jsou tím myšleny jednotlivé kategorie, oddělení, produkty, či uživatelské účty, způsob práce s nimi je pokaždé podobný. Odvíjí se od jejich úložiště.

V třívrstvých aplikacích bývají těmito úložišti databáze. A aplikační framework JBoss Seam má v sobě zabudovanou podporu návrhového vzoru DAO – který zaobaluje filozofii přístupu k databázovým objektům.

Tento přístup využívá třídy EntityHome, která je součástí knihoven Seamu. Tato třída slouží jako „prostředník“ v komunikaci s databází. Každá instance třídy odvíjené od EntityHome spravuje právě jednu instanci databázové entity, tím zastupuje jediný databázový záznam.

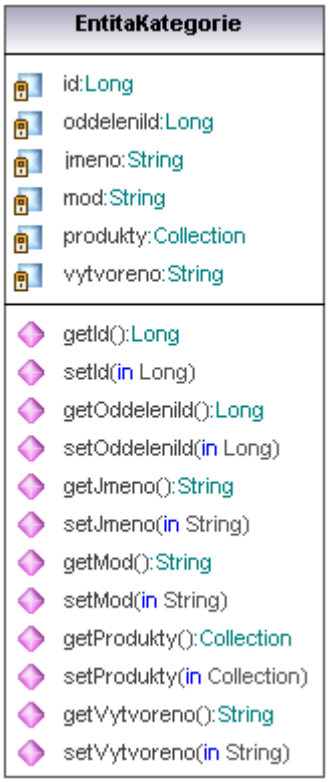
Více o schopnostech tříd odvíjených od EntityHome napoví obrázek 17, na kterém je zachycena třída EntityHome a její zasazení do hierarchie tříd Seamu.

Obr. 17: Diagram tříd pro EntityHome (obsahuje pouze část dostupných metod).
Zdroj: ALLEN, D. Seam In Action [1]

V rámci aplikace pak stačí pouze vytvořit vlastní „home objekty“, které definujeme jako potomky EntityHome. Nejlepší bude celý proces zmapovat na konkrétním příkladu.

V systému obchodního domu najdeme mnoho typických využití CRUD operací. Pro demonstraci tedy vyberu například manipulaci s kategoriemi jednotlivých oddělení v systému obchodního domu. Ostatní oblasti (jako přidávání oddělení, produktů do katalogu a další) jsou z funkčního hlediska velmi podobné. Filozofii jejich návrhu lze tedy snadno odvodit z předvedeného příkladu kategorií.

Prvním krokem pro vytvoření podpory CRUD operací je samozřejmě tvorba entitní třídy. Instance entity v aplikaci reprezentuje jeden konkrétní záznam z databáze. Obsahuje „set“ a „get“ metody k jednotlivým atributům, mapuje relace s ostatními entitami (tzn. relace mezi tabulkami v databázi). Entitní třída odpovídající záznamu v tabulce kategorie by mohla být definována tak, jak je tomu na obrázku 18.



Obr. 18: Návrhová třída entity reprezentující kategorii.

Zdroj: vlastní.

Podstatu entitní třídy samozřejmě netvoří pouze „set“ a „get“ metody. Aby byla třída

entitou, je nutné jí celou namapovat na databázový zdroj a její atributy na jednotlivé databázové sloupce, popř. správně definovat jednotlivé relace s ostatními tabulkami. Toho lze dosáhnout za pomocí Java anotací.

```
@Entity
@Table(name="KATEGORIE")
public class EntitaKategorie
{
      @Id @GeneratedValue
      @Column(name=„ID")
      private Long id;
      @Column(name=„ODDELENI_ID")
      private Long oddeleniId;
      @Column(name=„NAME")
      private String name;
      @Column(name=„MOD")
      private String mod;
      @Column(name=„VYTVORENO")
      private String vytvoreno;
      @OneToMany(mappedBy="KategorieId")
      private Collection produkty;
      ...
}
```

Tvořit takové entitní třídy ručně je samozřejmě hodně obtížné. Nesplést se při tvorbě jednotlivých vazeb přenášených z databázových závislostí je opravdu oříšek. Naštěstí ale dnešní dostupné vývojové Java nástroje (jako Netbeans, Eclipse) umí automaticky generovat entitní třídy přímo z databázových relací. Tyto vygenerované kostry již pak stačí v případě potřeby pouze upravovat, což je ale úkon mnohem snazší a průhlednější, než jejich samotná tvorba.

Ve chvíli, kdy jsou entity připravené, můžeme přistoupit k tvorbě konkrétního Home objektu pro práci s kategoriemi. Vytvoříme jej odvozením od objektu EntityHome.

```
@Name(„kategorieHome“)
public class KategorieHome extends EntityHome<EntitaKategorie>
{
...
}
```

V rámci domovského objektu kategorií pak můžeme v případě nutnosti upravit metody odvozené od rodiče – EntityHome. Tyto zděděné metody zabezpečují právě ony CRUD operace. Pokud se rozhodneme využít obecných metod definovaných v rodiči, nic tomu nebrání! Dalo by se tedy konstatovat, že k vytvoření funkční CRUD aplikace stačí pouze vytvořit holého potomka třídy EntityHome a tím veškerá ostatní práce končí.

V praxi je to ale přeci jen trochu jinak. Databázové záznamy nejednou uchovávají hodnoty, které je vhodné ve správnou chvíli generovat automaticky. Vezměme například v úvahu hodnotu sloupce VYTVORENO v databázové tabulce kategorie. Tato hodnota by měla být plněna automaticky aktuálním časem ve chvíli, kdy je objekt vytvořen. Je nesmysl vyžadovat hodnotu tohoto pole po uživateli.

Můžeme tedy v potomkovi – třídě KategorieHome přepsat metodu persist, která je volána ve chvíli, kdy se pokoušíme do databáze uložit nový (doposud neexistující) objekt.

```
protected String persist()
{
      EntitaKategorie kategorie = getInstance();
      kategorie.setVytvoreno(currentDate);
      setInstance(kategorie);
      return super.persist();
}
```

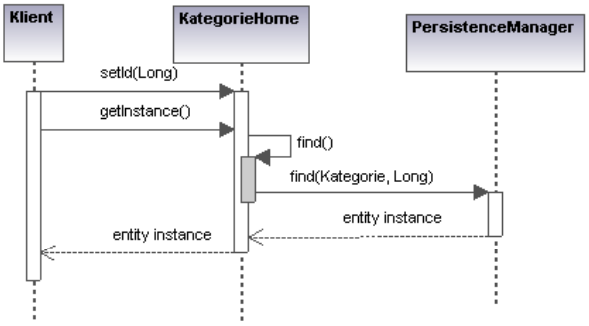
A to už je opravdu všechno! Posledním krokem je vytvořit přijatelné uživatelské rozhranní, přes které bude moci osoba oprávněná zakládat, rušit a editovat jednotlivé kategorie. EntityHome objekt nám dokonce umožňuje CRUD operace sjednotit do jednoho formuláře

– a to díky metodě isManaged(). Tato metoda totiž provádí jednoduchou kontrolu, zdali je instance entity již v modelu obsažená. Pokud není – zakládáme nový objekt, pokud je – aktualizujeme již existující objekt. Díky JSF a atributu rendered, tak docílíme univerzálního formuláře velmi jednoduše.

```
<h:form>
      <h:outputLabel value="Jméno kategorie: "/>
      <h:inputText value="#{kategorieHome.instance.jmeno}" />
      <h:commandButton rendered="#{kategorieHome.instance.managed}"
            value="Aktualizuj kategorii"
            action="#{kategorieHome.update}" />
      <h:commandButton rendered="#{!kategorieHome.instance.managed}"
            value="Založ novou kategorii"
            action="#{kategorieHome.persist}" />
      <h:commandButton rendered="#{kategorieHome.instance.managed}"
            value="Zruš kategorii"
            action="#{kategorieHome.delete}" />
</h:form>
```

Pokud již objekt existuje (metoda isManaged vrací true), uvidí uživatel pouze tlačítka „Aktualizuj kategorii“ a „Zruš kategorii“. Pokud naopak metoda vrací false, znamená to, že objekt doposud neexistuje. Jedná se tedy o proces založení nové kategorie a uživatel v tomto případě vidí pouze tlačítko „Založ novou kategorii“.

Podívejme se ještě podrobněji na to, jak probíhá asi nejčastější požadavek klienta na home objekt – tj. žádost o navrácení instance entity. Sekvenční diagram, který situaci popisuje, je zachycen na obrázku 19.



Obr. 19: Sekvenční diagram, který ukazuje, jak Home objekt vrací instanci entity. Zdroj: ALLEN, D. Seam In Action [1]

Klíčovou roli v tomto procesu hraje ID objektu. Není to žádné jiné ID, než databázové pole identifikované v entitní třídě anotací @Id. Právě na základě mapování ID pak dojde k napojení správné instance na home objekt.

3.4.2 Bezpečnost, autentizace a autorizace

V rámci aplikace, využívající vlastností frameworku JBoss Seam, probíhá vývoj autentizačního mechanizmu ve třech krocích:

1. Zapnutí autentizace správnou konfigurací autentizační metody.
2. Ověření identity uživatele v definované autentizační metodě.
3. Vytvoření JSF formuláře sloužícího pro přihlášení uživatelů.

Autentizační metoda se konfiguruje v konfiguračním souboru components.xml. A to za použití následujícího fragmentu kódu.

```
<security:identity
      authenticate-method="#{authenticationManager.authenticate}" />
```

Úkolem autentizační metody je prokázat identitu uživatele. Nejčastěji se tak děje za pomocí kontroly dvojice uživatelské jméno a heslo. Tato kontrola v rámci metody

neprobíhá automaticky – je nutné ji naprogramovat. Pokud kontrola proběhne úspěšně, měla by metoda vracet hodnotu true, pokud neúspěšně, tak naopak false.

A jaká je tedy klíčová role Seamu v tomto procesu a kde jsou výhody plynoucí z využití autentizační metody? Ty vznikají ve chvíli, kdy jednotlivá pole a akci samotnou v přihlašovacím formuláři připojíme k zabudované komponentě Seamu přístupné pod jménem identity (instance třídy Identity).

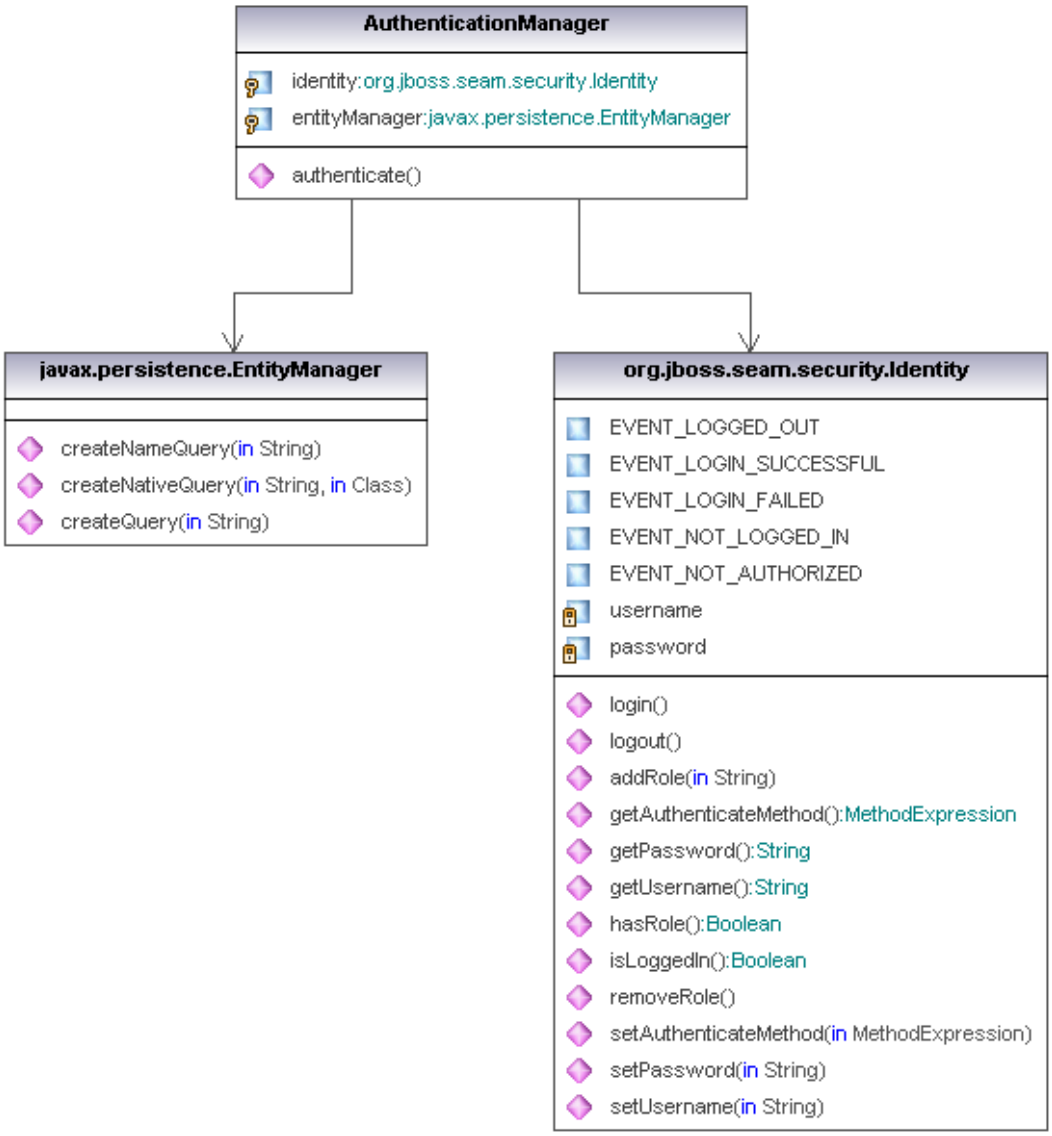
```
<h:form>
      <h:panelGrid columns="2">
            <h:outputLabel value="Jméno:" />
            <h:inputText value="#{identity.username}" />
            <h:outputLabel value="Heslo: " />
            <h:inputSecret value="#{identity.password}" />
            <h:commandButton value="Login" action="#{identity.login}"/>
      </h:panelGrid>
</h:form>
```

Asociační vazbu tříd AuthenticationManager, Identity a EntityManager (pro přístup k databázi) ukazuje diagram tříd na obrázku 20. Tuto vazbu v programovém kódu třídy AuthenticationManager vyjádříme následovně.

```
@Name(„authenticationManager")
public class AuthenticationManager()
{
      @In Identity identity;
      @In EntityManager entityManager;

      public boolean authenticate()
      {
            Uzivatel uzivatel = (Uzivatel)EntityManager.createQuery(
                  „select u from User u where
                  u.username=#{identity.username}).getSingleResult();
            ... kontrola uživatelského jména a hesla ...
      }
}
```

Povšimněte si důležité vlastnosti – v rámci dotazů volaných v prostředí Java kódu lze využívat expression language. A přes něj se jednoduše dostat až k hodnotě uživatelem zadaného uživatelského jména uloženého v zabudovaném objektu Seamu identity (#{identity.username}).



Obr. 20: Asociace mezi objekty AuthenticationManager a Identity (diagram obsahuje pouze výčet základních metod a atributů u tříd EntityManager a Identity).

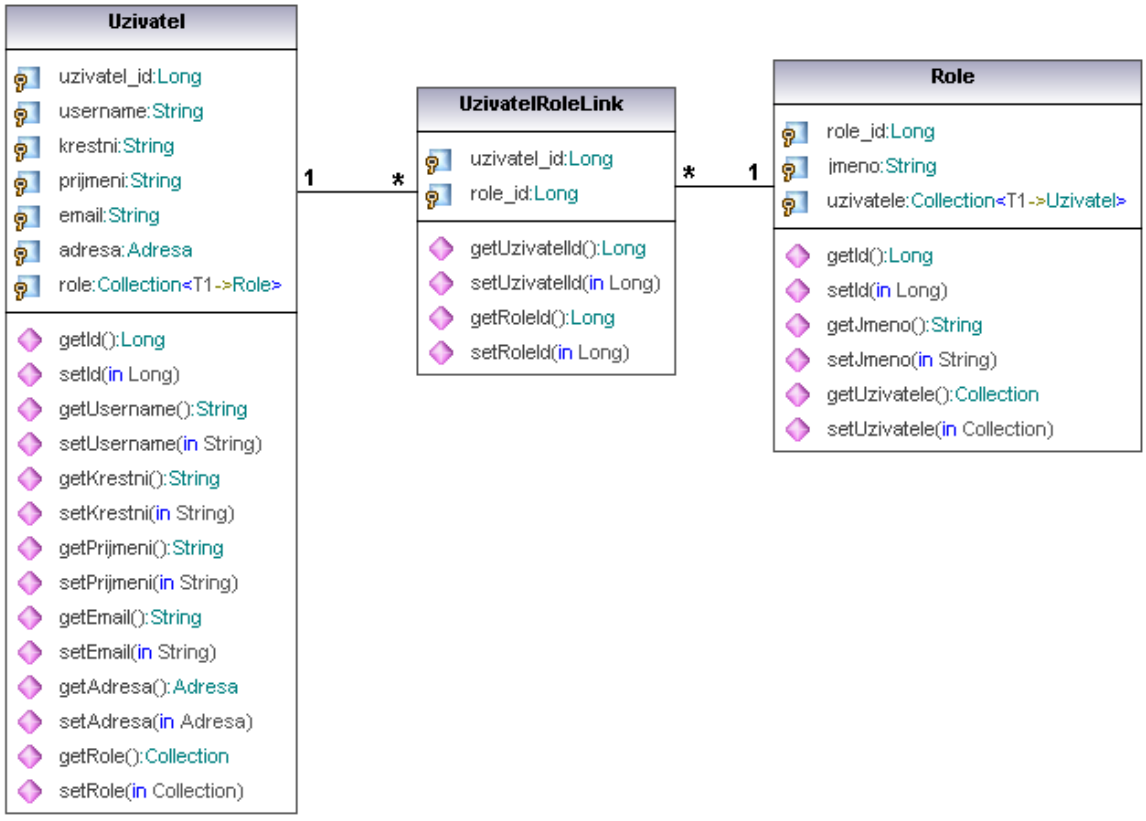
Zdroj: vlastní.

Jednotlivé stránky, či jejich komponenty, lze díky vlastnostem objektu identity zabezpečit proti neautorizovanému přístupu.

Zabezpečení na úrovni stránek lze zajistit v konfiguračním souboru pages.xml, kde pomocí atributu login-required specifikujeme, zdali je možné na stránky odpovídajícím kritériu view-id přistupovat anonymně, či nikoli.

Pokud chceme naopak zabezpečit jednotlivé komponenty, nabízí se využít atributu rendered a v rámci řetězce expression language volat metodu identity.isLoggedIn().

Mimo autentizace komponenta identity řeší i problematiku autorizace uživatele. Pomocí metody addRole objektu Identity (viz obr. 20) totiž můžeme uživateli v rámci autorizace přidat role, které mu v systému náleží. Abychom tuto problematiku správně pochopili, podívejme se na vztah entit zastupujících záznamy z databázových tabulek UZIVALEL a ROLE.



Obr. 21: Návrhové třídy Uzivatel a Role a relace mezi nimi.

Zdroj: vlastní.

Jak je z obrázku 21 patrné, platí mezi entitami vztah M:N. Tento druh relace nelze řešit přímo, je proto potřeba vytvořit linkovací tabulku UZIVATEL_ROLE_LINK, kterou v diagramu reprezentuje entita UzivatelRoleLink.

Pro úspěšnou autorizaci uživatele je nutné správně načíst role spojené s přihlašovaným uživatelem. Můžeme tak učinit v nitru autentizační metody a využít metody addRole objektu identity.

```
public boolean authenticate()
{
      Uzivatel uzivatel = (Uzivatel)EntityManager.createQuery(„select u
            from User u where
            u.username=#{identity.username}).getSingleResult();
      ... kontrola uživatelského jména a hesla ...
      for (Role role : uzivatel.getRole())
      {
            identity.addRole(role.getJmeno());
      }
}
```

Samotná ochrana proti nepovolenému přístupu pak může probíhat na dvou úrovních.

1. V aplikační vrstvě,
2. nebo v prezentační vrstvě.

V prezentační vrstvě můžeme opět využít atribut rendered definovaný u každé JSF komponenty. Protože hodnotou tohoto atributu může být libovolný expression language, který vrací buď true, nebo false, nic nebrání volat přímo komponentu identity a její metodu hasRole (viz obrázek 20). V rámci systému obchodního domu tuto funkcionalitu využijeme například nad vstupem do administrátorského rozhraní. Do této části systému by měli mít přístup pouze zaměstnanci, nikoli pouze zákazníci obchodního domu. Takový atribut rendered by mohl být u příslušného prvku definován následovně:

```
< … rendered="#{identity.hasRole('manazer_objednavek') or
      identity.hasRole('spravce_oddeleni')} or
      identity.hasRole('spravce_obchodniho_domu')" />
```

Kontrolované názvy rolí musí souhlasit s identifikátorem role, který jsme objektu identity přiřadili v rámci autentizační metody pomocí metody addRole – v tomto konkrétním případě se musí shodovat se jménem role v databázi. V některých publikacích je také možné vidět zkrácený výraz s:hasRole (namísto identity.hasRole). Oba výrazy jsou ale z hlediska funkcionality naprosto totožné.

Druhou možností, jak kontrolovat oprávněnost přístupu, je kontrola v rámci aplikační vrstvy. K tomuto účelu se využívá Seam anotace @Restrict, které jako String parametr předáváme expression výraz. Pokud tedy požadujeme, aby určitá operace byla vázána na přítomnost určité role, můžeme v rámci parametru předat opět expression language s voláním metody hasRole. Můžeme tak zabezpečit například metody v rámci home objektu reprezentujícího oddělení obchodního domu. Nová oddělení totiž může zakládat pouze správce celého obchodního domu.

```
public class OddeleniHome extends EntityHome<EntitaOddeleni>
{
      …
      @Restrict("#{identity.hasRole('spravce_obchodniho_domu')}")
      public String persist()
      {
            …
            return super.persist();
      }
}
```

Další problém související se zabezpečením aplikací je otázka, kdy využít protokol HTTP a kdy HTTPS. Elektronický obchodní dům můžeme i podle toho rozdělit do několika částí. Některé být zabezpečené nemusí, jiné naopak ano.

Například již výše zmiňované administrační rozhraní. To by zcela nepochybně mělo běžet pod zabezpečeným protokolem. Jiné příklady, které by bylo dobré zabezpečit, je přihlašovací mechanizmus, či realizace objednávky.

Aplikační framework Jboss Seam změnu protokolů podporuje velmi průhledně – a to v konfiguračním souboru pages.xml. V něm pomocí view-id atributu definujeme stránky, kterých se daná sekce pravidel týká a pomocí atributu schneme nastavíme příslušné schéma. Vezmeme-li například za svůj předpoklad, že všechny stránky administrace se budou nacházet ve složce /admin a vše související s přihlášením zase ve složce /login, bude fragment kódu pages.xml vypadat následovně.

```
<page view-id=”/admin/*” scheme=”https” />
<page view-id=”/login/*” scheme=”https” />
```

Jednotlivým protokolům pak pouze stačí v konfiguračním souboru components.xml nadefinovat správné porty.

```
<navigation:pages http-port=“8080“ http-port=“8443“ />
```

3.4.3 Využití kontextů Seamu

Další důležitou oblastí je rozdělení modulů systému obchodního domu do jednotlivých kontextů platnosti, které nabízí JBoss Seam. Důležité je hlavně správně specifikovat konverzace.

Jak již bylo řečeno v teoretické části, konverzace tvoří logický mezikrok mezi kontextem Event (zastupuje v Seamu kontext platnosti HTTP požadavku) a platností na úrovni session. Kontext Conversation by měl být využíván všude tam, kde se pracuje s nějakými průvodci (v příkladu obchodního domu např. registrace uživatele atd.), které vyžadují platnost proměnných přes několik požadavků.

Proces registrace uživatele z hlediska platnosti konverzace ukazuje jednoduchý stavový diagram, zachycený na obrázku 22. Proces se skládá ze dvou základních kroků.

1. Zadání osobních dat – uživatelského jména, hesla, příjmení atd.
2. Zadání doručovací adresy.

Nová konverzace začíná ve chvíli, kdy uživatel zadá povel k nové registraci, je platná při zadávání osobních dat i při přechodu k zadávání adresy. Končí, když uživatel potvrdí zadané údaje a adresa je v systému propojena s uživatelem.

Obr. 22: Stavový diagram zachycující proces registrace.
Zdroj: vlastní

V programovém kódu tohoto chování docílíme za pomocí Java anotací, které nabízí Seam. Nabízí se je využít v rámci home objektu uživatele – uzivatelHome. Protože konverzace by měla začít ve chvíli, kdy se vytvoří prázdná instance entity zastupují uživatele – můžeme anotaci @Begin připojit k metodě createInstance(). Konverzaci naopak ukončíme anotací

@End umístěnou nad metodou persist(), která je volána ve chvíli, kdy nově vzniklého uživatele (s připojenou adresou) ukládáme do databáze.

Stejně tak je vhodné konverzaci založit pro manipulaci s existujícími daty – např. pro zakládání a editaci nových kategorií, produktů, oddělení. Protože problematika manipulace s jednotlivými záznamy byla řešena multifunkčním formulářem detailu (za využití metody home objektu isManaged bylo rozpoznáváno, jaké operace jsou přístupné), můžeme si zjednodušit práci a identifikovat právě stránku s tímto formulářem jako tu, která automaticky nastartuje dlouhodobou konverzaci. Toho docílíme jednoduchou konfigurací v souboru pages.xml.

Předpokládejme, že všechny předlohy formulářů detailu jsou umístěny ve složce /admin/details. Pak pro automatický start konverzace při přístupu na stránku musíme uvést následující blok kódu. V rámci expression laguage v parametru if definujeme podmínku, která zaručí, že požadavek bude vykonán pouze tehdy, pokud ještě nebyla žádná dlouhodobá konverzace aktivována. Tím zamezíme tomu, aby se nevytvářely konkurenční konverzace, pokud by uživatel vstupoval na tyto stránky již v rámci nějaké otevřené dlouhodobé konverzace (např. zakládání nového zaměstnance atd.).

```
<pages view-id="/admin/details/*">
      <begin-conversation if="#{!conversation.isLongRunning()}" />
</pages>
```

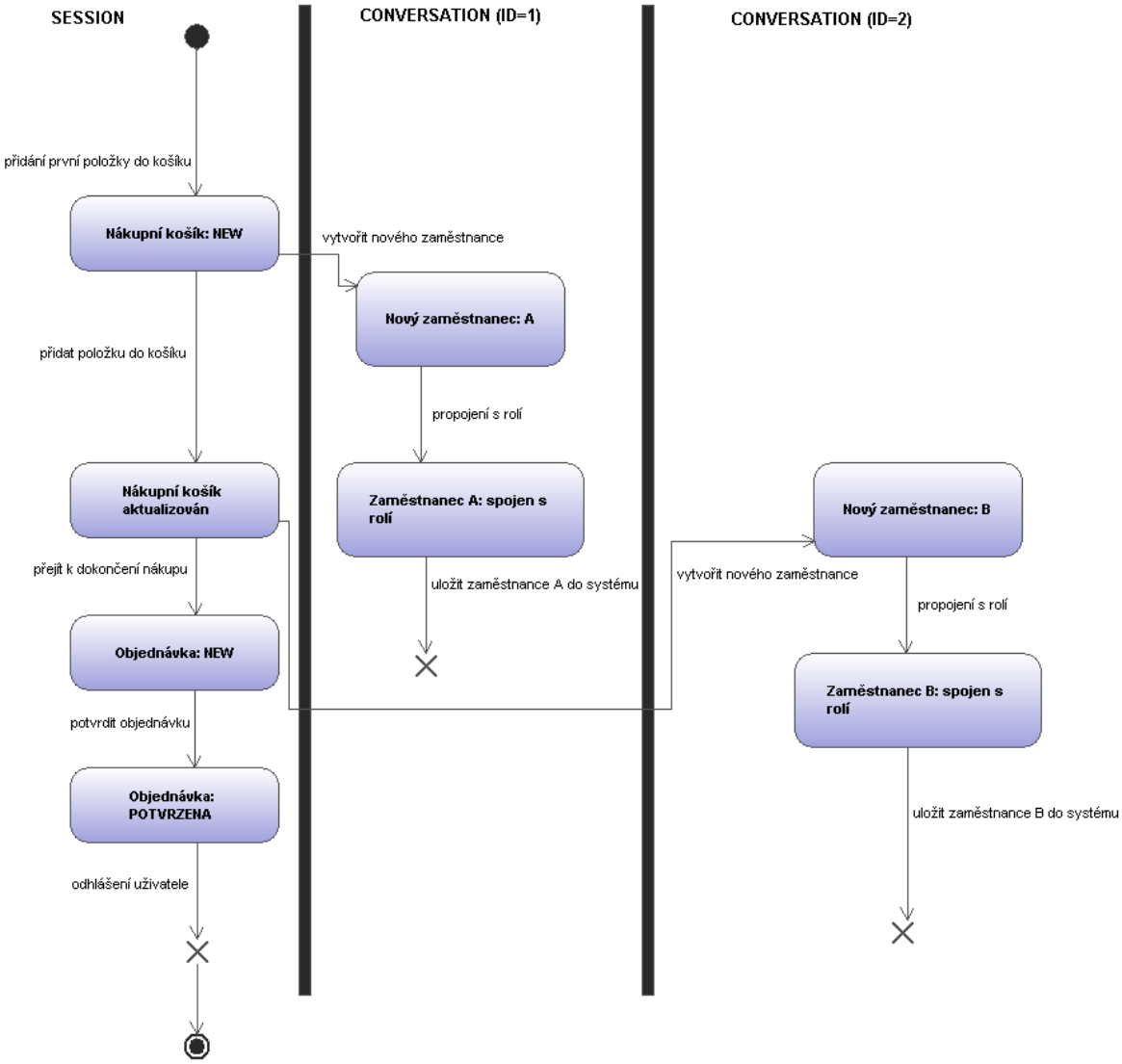
Jedna z vlastností konverzace je totiž schopnost manipulovat nezávisle s jednotlivými záložkami v rámci prohlížečů. Správná definice konverzací umožní tedy uživateli pracovat nezávisle na několika problémech najednou, aniž by se tyto oblasti dostávaly do konfliktu. Každá konverzace má totiž své jednoznačné identifikační číslo, pomocí kterého je jednoznačně identifikována.

Zjednodušeně se dá říci, že konverzace by měla odpovídat jednotlivým případům užití. Tato definice ale přesto není zcela univerzální a je třeba se zamyslet nad tím, jestli by přeci

jen některé případy užití neměly spadat pod kontext Session.

Na příkladu obchodního domu to může být třeba samotný nákup. Nákup začíná ve chvíli, kdy uživatel přidá první položku do nákupního košíku. Je nesmysl umožnit jednomu uživatelskému účtu zakládat v různých záložkách nezávislé nákupy. Lze tedy usoudit, že obsah nákupního košíku je vhodné sdílet v rámci kontextu Session (ve kterém drží svou platnost i objekt identity).

Celkový pohled na problematiku kontextů Session a Conversation ukazuje alespoň z části obrázek 23. Z něj jasně vyplývá, že v rámci různých konverzací lze provádět stejné úkony, aniž by docházelo ke konfliktům (na obrázku se jedná o souběžné zakládání dvou nových zaměstnanců jedním uživatelem realizováno ve dvou otevřených záložkách webového prohlížeče). Tento uživatel má ale zároveň otevřený proces nakupování. Ten je však sdílen v rámci kontextu Session – je tedy sdílen mezi všemi otevřenými záložkami.



Obr. 23: Stavový diagram pro uživatele, který zároveň nakupuje a v rámci dvou otevřených konverzací zakládá dva nové zaměstnance.

Zdroj: vlastní

3.4.4 Návrh prezentační vrstvy

Komponenty čistého JSF jsou poměrně strohé. Nemají v sobě zabudovánu podporu moderních webových technologií, jako je například Ajax (Asynchronous JavaScript and XML) pro částečné překreslování obsahu stránky. Stejně tak datové tabulky v JSF nepodporují filtrování, řazení atd. Všechny tyto vlastnosti je v případě využití čistého JSF nutné programovat ručně.

Existuje ale ještě další možnost – a tou je využít některé z implementací JSF, které jsou volně dostupné. V rámci knihoven Seamu je dokonce přímo k dispozici komponentová knihovna JBoss Richfaces. V aplikaci obchodního domu tedy můžeme volně využívat bohaté komponenty, které nabízí.

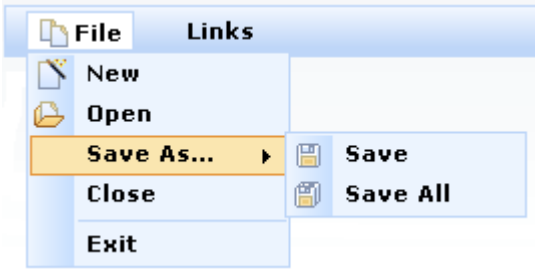
Například komponenta extendedDataTable nabízí již předpracované filtrování a je velmi snadné jej využít pro uživatelské vyhledávání. A to s minimem přidaného zdrojového kódu.

Flag	State Name ⇕	State Capital ⇕	Time Zone	
	Al			
	Alabama	Montgomery	GMT-6	
	Alaska	Juneau	GMT-9	

Obr. 24: Komponenta ExtendedDataTable.

Zdroj: http://livedemo.exadel.com/richfaces-demo/richfaces/

Dále se nabízí využít možností po grafické stránce velmi povedených menu. Například komponenta dropDownMenu zase simuluje v aplikaci podobné menu, známé z operačního systému (viz obr. 25).



Obr. 25: Komponenta dropDownMenu

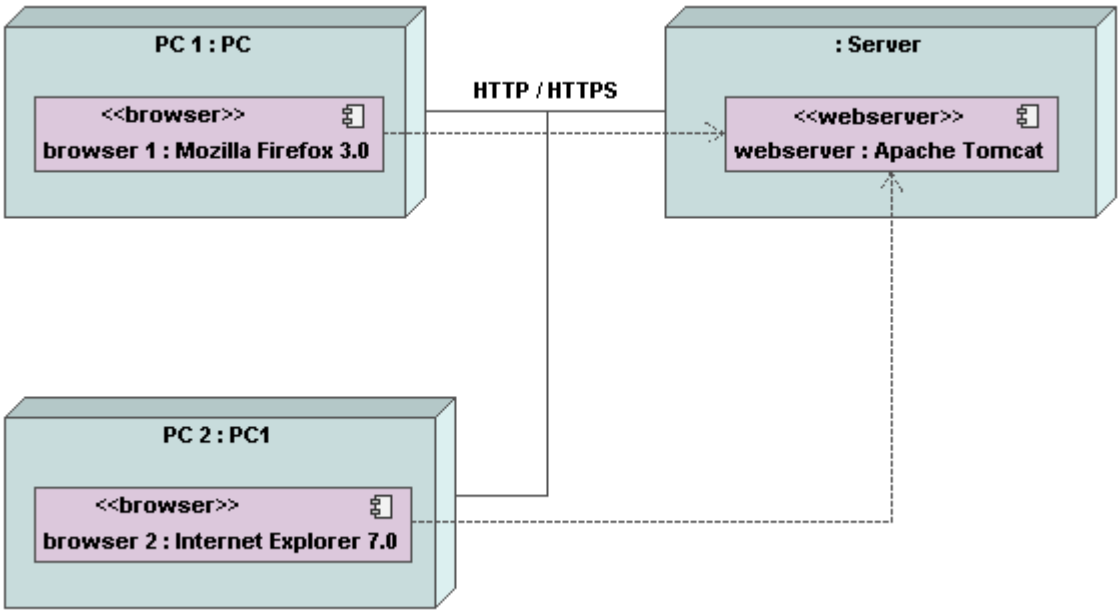
Zdroj: http://livedemo.exadel.com/richfaces-demo/richfaces/

Toho všeho docílíme pouze správným přidáním komponenty do zdrojového kódu XHTML souboru.

3.5 Pracovní postup: Implementace

V rámci implementace obchodního domu se budu zabývat především problematikou nasazení systému obchodního domu do prostředí internetu. Celý systém obchodního domu musí samozřejmě běžet na serveru s podporou Javy.

Takovým serverem je velmi často server Apache Tomcat. Uživatelé pak komunikují se serverem přes HTTP (nebo HTTPS) požadavky ze svých domácích stanic a využívají přitom libovolné internetové prohlížeče v roli tenkého klienta.



Obr. 26: Diagram nasazení systému obchodního domu.

Zdroj: vlastní.

3.6 Ekonomická analýza řešení

Podívejme se nyní na to, co využití moderních technologií sloučených v integračním frameworku Seam přinese společnostem, které konkréntní e-business aplikace (např. prezentovaný systém obchodního domu) vyvíjí. Jaké ekonomické výhody jim poskytne?

V prvé řadě využívání moderní technologie ztělesněné frameworkem Seam uspoří celkový čas vývoje aplikace. Urychlí celý její životní cyklus. Programátoři jsou schopni snadno a rychle tvořit funkčně bohatou a v případě využití implementací JSF (např. Richfaces) i po designové stránce poměrně povedenou aplikaci. Ušetří si mnoho řádků programového kódu (a spoustu hodin práce). Tento ušetřený čas lze samozřejmě využít k plnění dalších úkolů a tak celý proces vývoje aplikace v rámci společnosti zproduktivnit.

Mnohem jednodušším a efektivnějším se také stává i zapojení nových členů do vývojového týmu. Protože v okamžiku, kdy člen týmu vstřebá teoretické znalosti o frameworku, JSF a třívrstvé architektuře za pomocí Facelets, stává se pro něj aplikace rychle průhledná. Zařazení nové posily vývojového týmu se tedy mnohem rychleji mění v návratnou investici, než tomu v případě rozšiřování týmu u rozběhlých aplikací založených například na servetech a JSP.

Další nepochybnou výhodou, kterou přináší využívání nových technologií, je pružnost takové aplikace. Systém tvořený na základech Seamu a technologie Facelets je totiž opravdovou třívrstvou aplikací. A takovou aplikaci lze velmi snadno a rychle upravovat v závislosti na nových požadavcích a tím získávat body navíc ve srovnání s konkurenčními projekty.

4 Závěr

V současné době se v Javě vyvíjejí internetové aplikace za pomocí několika dostupných technologií. Prvním cílem této diplomové práce bylo zmapovat je a na základě jak teoretických znalostí, tak i mnou načerpaných praktických zkušeností, je porovnat. Následně pak některé z nich označit za progresivní, jiné naopak pomalu považovat za zastaralé (Java servlety, JSP).

Na základě argumentů podložených teoretickými i praktickými znalostmi z tohoto porovnání vzešel jako vítěz aplikační framework JBoss Seam, který šikovně zaobaluje ostatní hojně využívané technologie (JSF, JBoss Richfaces, Facelets, Hibernate), jejichž funkčnost koordinuje a vhodně doplňuje.

Ale i přesto, že jsem rozdělila technologie na progresivnější a ustupující, neznamená to, že bych např. JSP a servlety zcela zavrhovala. Vývoj jakékoli rozsáhlejší e-business aplikace je totiž zároveň „během na dlouhou trať“. Spousta v současnosti hojně využívaných a rozsáhlých aplikací, které hrají ve své oblasti nezpochybnitelný prim, dnes slaví mnohaletá výročí. A to ačkoli vznikaly na těchto „zastaralých“ technologiích, plní své úkoly velmi dobře, nejednou mnohem lépe, než jejich moderní, ale zato mladší kolegové založené na progresivnější programátorské vlně.

Žádná aplikace by se neměla stát obětí války na poli technologie. Všechny dnes „zastaralé“ technologie byly také kdysi moderní a progresivní. A dá se s jistotou říci, že vývoj bude i nadále pokračovat a pravděpodobně i dnes progresivní technologie jednoho dne „zastarají“. Pokud by se aplikace měla dynamickému vývoji na poli technologií neustále podřizovat, znamenalo by to jediné. Neustálé přepisování programového kódu a její nová a nová nasazování. Tak by však dříve nebo později došlo ke zpomalení jejího rozvoje a tím plynoucím ztrátám v konkurenčním boji. Ale právě schopnost pružně vyhovět dynamicky se měnícím požadavkům vytváří ve výsledku podmínky pro rozhodnutí, zda je aplikace úspěšná, či nikoli. Technologie samotná je pomocník vývojářů, konečné zákazníky ale podrobnosti o ní příliš nezajímají.

Dalším cílem práce bylo prokázat vlastní schopnosti v oblasti objektově orientovaného vývoje systémů. Na základě praktických zkušeností a za použití objektové metodiky UP jsem předložila postup tvorby vzorové aplikace – elektronického obchodního domu. Od prvního kroku – modelování požadavků. Přes objektovou analýzu až k srdci samotné problematiky – návrhu. Cílem této části byl vývoj návrhového modelu založeného na analýze, avšak s přihlédnutím k realizaci systému za využití integračního frameworku JBoss Seam. Ukázala jsem, jak lze jednotlivé zabudované vlastnosti Seamu využít ve prospěch vývojářů, i samotné aplikace.

Dalším cílem, kterého bylo v rámci diplomové práce dosaženo, bylo pak prokázat přínos nových technologií na poli vývoje softwaru, včetně prokázání ekonomických přínosů moderních řešení e-business aplikací. Využívání moderních technologií (jako je JBoss Seam) totiž zřetelně šetří čas nutný k provádění jednotlivých programátorských úkonů. Umožňuje pružně reagovat na uživatelské požadavky, protože aplikace psaná v duchu Seamu šetří spousty řádek programového kódu a dbá na dodržování třívrstvé architektury aplikace. Tak tedy i nepřímo sama technologie pomáhá přinášet kladné body v konkurenčním boji.

V rámci kapitoly Modelování systémového obchodního domu jsem zase pokusila potvrdit své schopnosti a praktické zkušenosti s vývojem objektově orientovaných systémů, které jsem v průběhu studií a své tříleté praxe v oboru programování v jazyku Java načerpala. Na úplný závěr si troufám tvrdit, že systém obchodního domu, odvozený z této diplomové práce, by byl provozuschopný a po technologické stránce by obstál.

5 Citace

[1] ALLEN, D. *Seam in action*. 1st ed. Greenwich: Manning, 2008. 589 pgs.

ISBN 1933988401.

[2] ARLOW, J. a NEUSTADT, I. *UML a unifikovaný proces vývoje aplikací*. 1. vyd. Brno: Computer Press, 2005. 387 s. ISBN 80-7226-947-X.

[3] BOLLINGER, G. JSP*: Podrobný průvodce začínajícího tvůrce*. 1. vyd. Praha: Grada Publishing, 2003. 418 s. ISBN 80-247-0340-8

[4] BUCHALCEVOVÁ, A. *Metody vývoje a údržby informačních systémů*. 1. vyd. Praha: Grada Publishing, 2005. 163 s. ISBN ISBN 80-247-1075-7.

[5] GEARY, D. and HORSTMANN, C. *Core JavaServer Faces*. 2nd ed. New York: Prentice Hall, 2007. 723 pgs. ISBN 978-0-13-173886-7.

[6] HALL, M. *Java servlety a stránky JSP*. 1. vyd. Praha: Neocortex, 2001. 586 s. ISBN 80-86330-06-0.

[7] HIGHTOWER, R. *JSF for nonbelievers: JSF conversion and validation*. [online] 2008. [cit. 2008-09-12]. Dostupný z WWW:
<http://www.ibm.com/developerworks/java/library/j-jsf3/ >

[8] HIGHTOWER, R. *JSF for nonbelievers: The JSF application Life cycle.* [online] 2008. [cit. 2008-09-12]. Dostupný z WWW:
<http://www.ibm.com/developerworks/java/library/j-jsf2/ >

[9] HIGHTOWER, R. *Facelets fits JSF like a glove.* [online] 2008. [cit. 2008-09-13]. Dostupný z WWW: < http://www.ibm.com/developerworks/java/library/j-facelets/>

[10] KUBÁSEK, L. *Představení aplikačního frameworku JBoss Seam*. 2008. [cit. 2008-09-13]. Dostupný z WWW: < http://www.kubasek.cz/blog/2008/02/17/predstaveni-aplikacniho-frameworku-jboss-seam/>

[11] PALETA, P. *Co programátory ve škole neučí*. 1. vyd. Brno: Computer Press, 2003.

337 s. ISBN 80-251-0073-1.

[12] TIOBE index [online] Eindhoven: TIOBE company, 2009 [cit. 2009-05-10]. Dostupný z WWW: <http://www.tiobe.com>

6 Bibliografie

[13] ALLEN, D. *Seamless JSF, Part 1: An application framework tailor-made for JSF*. [online] 2009. [cit. 2009-01-02]. Dostupný z WWW: <http://www.ibm.com/developerworks/java/library/j-seam1/>

[14] ALLEN, D. *Seamless JSF, Part 2: Conversation with Seam*. [online] 2009. [cit. 2009-01-02]. Dostupný z WWW:<http://www.ibm.com/developerworks/java/library/j-seam2/>

[15] HIGHTOWER, *R. JSF for nonbelievers: Clearing the FUD about JSF.* [online] 2008. [cit. 2008-09-12]. Dostupný z WWW: http://www.ibm.com/developerworks/java/library/j-jsf2/>

[16] HIGHTOWER, R. *JSF for nonbelievers: JSF Component Development* [online] 2008. [cit. 2008-09-12]. Dostupný z WWW: http://www.ibm.com/developerworks/java/library/j-jsf4/>

[17] PECINOVSKÝ, R. *Návrhové vzory*. 1. vyd. Brno: Computer Press, 2007. 527 s. ISBN 978-80-251-1582-4.

[18] SPELL, B. *Java Programujeme profesionálně*. 1. vyd. Praha: Computer Press, 2002. 1021 s. ISBN 80-7226-664-5.